Object Oriented Programming using C#

Simon Kendal

Simon Kendal

Object Oriented Programming using C#

Object Oriented Programming using C#

ISBN 978-87-7681-814-2

Contents

"To Janice and Cara: jewels forged in heaven, shining here on earth!"

Simon Kendal

Foreword

This book aims to instil the reader with an understanding of the Object Oriented approach to programming and aims to develop some practical skills along the way. These practical skills will be developed by small exercises that the reader will be invited to undertake and the feedback that will be provided.

The concepts that will be explained and skills developed are in common use among programmers using many modern object oriented languages and are thus transferrable from one language to another. However for practical purposes these concepts are explored and demonstrated using the C# (pronounced C sharp) programming language.

While the C# programming language is used to highlight and demonstrate the application of fundamental object oriented principles and modelling techniques this book is not an introduction to C# programming. The reader will be expected to have an understanding of basic programming concepts and their implementation in C# (inc. the use of loops, selection statements, performing calculations, arrays, data types and a basic understanding of file handling).

This text is designed not as a theoretical textbook but as a learning tool to aid in understanding theoretical concepts and learning the practical skills required to implement these. To this end each chapter will incorporate small exercises with solutions and feedback provided.

At the end of the book one larger case study will be described – this will be used to illustrate the application of the techniques explored in the earlier chapters. This case study will culminate in the development of a complete C# program that can be downloaded with this book.

1 An Introduction to Object Orientated Programming

Introduction

This chapter will discuss different programming paradigms and the advantages of the Object Oriented approach to software development and modelling. The concepts on which object orientation depend (abstraction, encapsulation, inheritance and polymorphism) will be explained.

Objectives

By the end of this chapter you will be able to….

- Explain what Object Oriented Programming is,
- Describe the benefits of the Object Oriented programming approach and
- Understand the basic concepts of abstraction, encapsulation, generalisation and polymorphism on which object oriented programming relies.
- Understand the reasons behind the development of the .NET framework and the role of the Common Language Runtime (CLR) engine.

All of these issues will be explored in much more detail in later chapters of this book.

This chapter consists of nine sections :-

1) A Brief History of Computing
2) Different Programming Paradigms
3) Why use the Object Oriented Paradigm?
4) Object Oriented Principles
5) What Exactly is Object Oriented Programming?
6) The Benefits of the Object Oriented Programming Approach.
7) Software Implementation
8) An Introduction to the .NET Framework
9) Summary

1.1 A Brief History of Computing

Computing is a constantly changing our world and our environment. In the 1960s large machines called mainframes were created to manage large volumes of data (numbers) efficiently. Bank account and payroll programs changed the way organisations worked and made parts of these organisations much more efficient. In the 1980s personal computers became common and changed the way many individuals worked. People started to own their own computers and many used word processors and spreadsheets applications (to write letters and to manage home accounts). In the 1990s email became common and the world wide web was born. These technologies revolutionised communications allowing individuals to publish information that could easily be accessed on a global scale. The ramifications of these new technologies are still not fully understood as society is adapting to opportunities of internet commerce, new social networking technologies (twitter, facebook, myspace, online gaming etc) and the challenges of internet related crime.

Just as new computing technologies are changing our world so too are new techniques and ideas changing the way we develop computer systems. In the 1950s the use machine code (unsophisticated, complex and machine specific) languages were common.

In the 1960s high level languages, which made programming simpler, became common. However these led to the development of large complex programs that were difficult to manage and maintain.

In the 1970s the structured programming paradigm became the accepted standard for large complex computer programs. The structured programming paradigm proposed methods to logically structure the programs developed into separate smaller, more manageable components. Furthermore methods for analysing data were proposed that allowed large databases to be created that were efficient, preventing needless duplication of data and protected us against the risks associated with data becoming out of sync. However significant problems still persisted in a) understanding the systems we need to create and b) changing existing software as users requirements changed.

In the 1980s 'modular' languages, such as Modula-2 and ADA were developed that became the precursor to modern Object Oriented languages.

In the 1990s the Object Oriented paradigm and component-based software development ideas were developed and Object Oriented languages became the norm from 2000 onwards.

The object oriented paradigm is based on many of the ideas developed over the previous 30 years of abstraction, encapsulation, generalisation and polymorphism and led to the development of software components where the operation of the software and the data it operates on are modelled together. Proponents of the Object Oriented software development paradigm argue that this leads to the development of software components that can be re-used in different applications thus saving significant development time and cost savings but more importantly allow better software models to be produced that make systems more maintainable and easier to understand.

It should perhaps be noted that software development ideas are still evolving and new agile methods of working are being proposed and tested. Where these will lead us in 2020 and beyond remains to be seen.

1.2 Different Programming Paradigms

The structured programming paradigm proposed that programs could be developed in sensible blocks that make the program more understandable and easier to maintain.

Activity 1

Assume you undertake the following activities on a daily basis. Arrange this list into a sensible order then split this list into three blocks of related activities and give each block a heading to summarise the activities carried out in that block.

Get out of bed
Eat breakfast
Park the car
Get dressed
Get the car out of the garage
Drive to work
Find out what your boss wants you to do today
Feedback to the boss on today's results.
Do what the boss wants you to do

Feedback 1

You should have been able to organise these into groups of related activities and give each group a title that summarises those activities.

Get up :-
- Get out of bed
- Get dressed
- Eat breakfast

Go to Work :-
- Get the car out of the garage
- Drive to work
- Park the car

Do your job :-
- Find out what your boss wants you to do today
- Do what the boss wants you to do
- Feedback to the boss on today's results.

By structuring our list of instructions and considering the overall structure of the day (Get up, go to work, do your job) we can change and improve one section of the instructions without changing the other parts. For example we could improve the instructions for going to work….

- Listen to the local traffic and weather report
- Decide whether to go by bus or by car
- If going by car, get the car and drive to work.
- Else walk to the bus station and catch the bus

without worrying about any potential impact this may have on 'getting up' or 'doing your job'. In the same way structuring computer programs can make each part more understandable and make large programs easier to maintain.

The Object Oriented paradigms suggest we should model instructions in a computer program with the data they manipulate and store these as components together. One advantage of doing this is we get reusable software components.

Activity 2

Imagine a personal address book with some data stored about your friends
- Name,
- Address,
- Telephone Number.

List three things that you may do to this address book.

Next identify someone else who may use an identical address book for some purpose other than storing a list of friends.

Feedback 2

With an address book we would want to be able to perform the following actions :- find out details of a friend i.e. their telephone number, add an address to the address book and, of course, delete an address.

We can create a simple software component to store the data in the address book (i.e. list of names etc) and the operations, things we can do with the address book (i.e add address, find telephone number etc).

By creating a simple software component to store and manage addresses of friends we can reuse this in another software system i.e. it could be used by a business manager to store and find details of customers. It could also become part of a library system to be used by a librarian to store and retrieve details of the users of the library.

Thus in object oriented programming we can create re-usable software components (in this case an address book).

The Object Oriented paradigm builds upon and extends the ideas behind the structured programming paradigm of the 1970s.

1.3 Why use the Object Orientation Paradigm?

While we can focus our attention on the actual program code we are writing, whatever development methodology is adopted, it is not the creation of the code that is generally the source of most problems. Most problems arise from :-

- poor maintainability: the system is hard to understand and revise when, as is inevitable, requests for change arise.
- Statistics show 70% of the cost of software is not incurred during its initial development phase but is incurred during subsequent years as the software is amended to meet the ever changing needs of the organisation for which it was developed. For this reason it is essential that software engineers do everything possible to ensure that software is easy to maintain during the years after its initial creation.

The Object Oriented programming paradigm aims to help overcome these problems by helping with the analysis and design tasks during the initial software development phase (see chapter 6 for more details on this) and by ensuring software is robust and maintainable (see chapters 9 -11 for information on the support Object Orientation and C# provides for creating systems that are robust and maintainable).

1.4 Object Oriented Principles

Abstraction and encapsulation are fundamental principles that underlie the Object Oriented approach to software development. Abstraction allows us to consider complex ideas while ignoring irrelevant detail that would confuse us. Encapsulation allows us to focus on what something does without considering the complexities of how it works.

Activity 3 Consider your home and imagine you were going to swap your home for a week with a new friend.

Write down three essential things you would tell them about your home and that you would want to know about their home.

Now list three irrelevant details that you would not tell your friend.

Feedback 3

You presumably would tell them the address, give them a basic list of rooms and facilities (e.g. number of bedrooms) and tell them how to get in (i.e which key would operate the front door and how to switch off the burglar alarm (if you have one).

You would not tell them irrelevant details (such as the colour of the walls, seats etc) as this would overload them with useless information.

Abstraction allows us to consider the important high level details of your home, e.g. the address, without becoming bogged down in detail.

Activity 4 Consider your home and write down one item, such as a television, that you use on a daily basis (and briefly describe how you operate this item).

Now consider how difficult it would be to describe the internal components of this item and give full technical details of how it works.

Feedback 4

Describing how to operate a television is much easier than describing its internal components and explaining in detail exactly how it works. Most people do not even know all the components of the appliances they use or how they work – but this does not stop them from using appliances every day.

You may not know the technical details such as how the light switches are wired together and how they work internally but you can still switch the lights on and off in your home (and in any new building you enter).

Encapsulation allows us to consider what a light switch does, and how we operate it, without needing to worry about the technical detail of how it actually works.

Two other fundamental principles of Object Orientation are Generalization/specialization (which allows us to make use of inheritance) and polymorphism.

Generalisation allows us to consider general categories of objects which have common properties and then define specialised sub classes that inherit the properties of the general categories.

Activity 5 Consider the people who work in a hospital-list three common occupations of people you would expect to be employed there.

Now for each of these common occupations list two or three specific categories of staff.

Feedback 5 Depending upon your knowledge of the medical profession you may have listed three very general occupations (e.g. doctor, nurse, cleaner) or you may have listed more specific occupations such as radiologist, surgeon etc.

Whatever your initial list you probably would have been able to specify more specialised categories of these occupations e.g.

Doctor :-

Trainee doctor,
Junior doctor,
Surgeon,
Radiologist,
etc

Nurse :-

Triage nurse,
Midwife,
Ward sister

Cleaner :-

General cleaner
Cleaning supervisor

Now we have specified some general categories and some more specialised categories of staff we can consider the general things that are true for all doctors, all nurses etc.

Activity 6 Make one statement about doctors that you would consider to be true for all doctors and make one statement about surgeons that would **not** be true for all doctors.

Feedback 6 You could make a statement that all doctors have a knowledge of drugs, can diagnose medical conditions and can prescribe appropriate medication.

For surgeons you could say that they know how to use scalpels and other specialised pieces of equipment and they can perform operations.

According to our list above all surgeons are doctors and therefore still have a knowledge of medical conditions and can prescribe appropriate medication. However not all doctors are surgeons and therefore not all doctors can perform operations.

Whatever we specify as true for doctors is also true for trainee doctors, junior doctors etc – these specialised categories (or classes) can inherit the attributes and behaviours associated with the more general class of 'doctor'.

Generalisation / specialisation allow us to define general characteristics and operations of an object and allow us to create more specialised versions of this object. The specialised versions of this object will automatically inherit all of the characteristics of the more generalised object.

The final principle underlying Object Orientation is Polymorphism which is the ability to interact with a object as its generalized category regardless of its more specialised category.

Activity 7 Make one statement about how a hospital manager may interact with all doctors employed at their hospital irrespective of what type of doctor they are.

Feedback 7 You may have considered that a hospital manager could pay all doctors (presumably this will be done automatically at the end of every month) and could discipline any doctor guilty of misconduct – of course this would be true for other staff as well. More specifically a manager could check that a doctor's medical registration is still current. This would be something that management would need to do for all doctors irrespective of what their specialism is.

Furthermore if the hospital employed a new specialist doctor (e.g. a Neurologist), without knowing anything specific about this specialism, hospital management would still know that a) these staff needed to be paid and b) their medical registration must be checked. i.e. they are still doctors and need to be treated as such.

Using the same idea polymorphism allows computer systems to be extended, with new specialised objects being created, while allowing current parts of the system to interact with new object without concern for the specific properties of the new objects.

1.5 What Exactly is Object Oriented Programming?

Activity 8 Think of an object you possess. Describe its current state and list two or three things you can do with that object.

Feedback 8 You probably thought about an entirely physical object such as a watch, a pen, or a car.

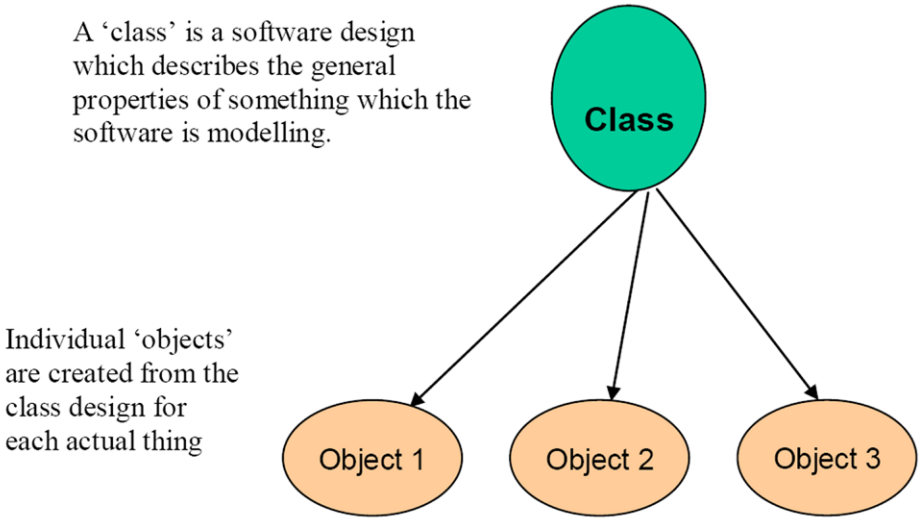
Objects have a current status. A watch has a time (represented internally by wheels and cogs or in an electronic component). A pen has a certain amount of ink in it and has its lid on or off. A car has a current speed and has a certain amount of fuel inside it.

Specific behaviour can also be associated with each object (things that you can do with it) :- a watch can be checked to find out its time, its time can also be set. A pen can be used to write with and a car can be, started, driven and stopped.

You can also think of other non physical things as objects :- such as a bank account. A bank account is not something that can be physically touched but intellectually we can consider a bank account to be an object. It also has a current status (the amount of money in it) and it also has behaviour associated with it (most obviously deposit money and withdraw money).

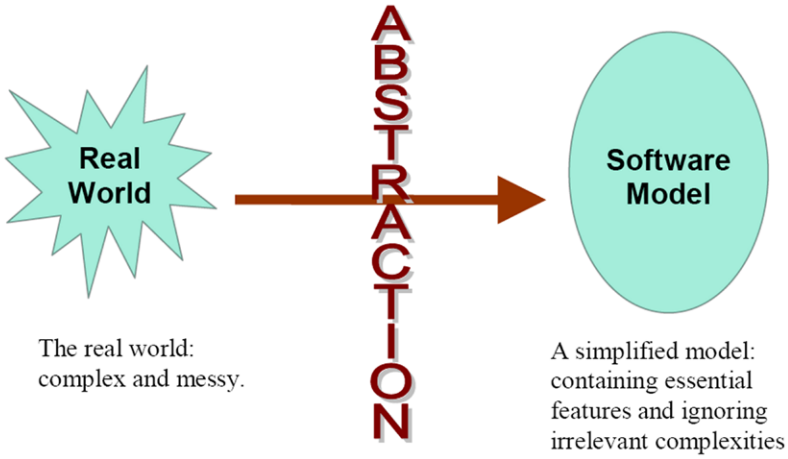
Object oriented programming it a method of programming that involves the creation of intellectuals objects that model a business problem we are trying to solve (e.g. a bank account, a bank customer and a bank manager – could all be objects in a computerised banking system). With each object we model the data associated with it (i.e. it status at any particular point in time) and the behaviour associated with it (what our computer program should allow that object to do).

In creating a object oriented program we define the properties of a class of objects (e.g. all bank accounts) and then create individual objects from this class (e.g. your bank account).



However deciding just what classes we should create in our system is not a trivial task as the real world is complex and messy (see chapter 6 for more advice on how to go about this). In essence we need to create an abstract model of the real world that focuses on the essential aspects of a problem and ignores irrelevant complexities. For example in the real a world bank account holders sometimes need to borrow money and occasionally their money may get stolen by a pick pocket. If we were to create a bank account system should we allow customers to borrow money? Should we acknowledge that their cash may get stolen and build in some method of them getting an immediate loan – or is this an irrelevant detail that would just add complexity to the system and provide no real benefit to the bank?

Using object oriented analysis and design techniques our job would be to look at the real world and come up with a simplified abstract model that we could turn into a computer system. How good our final system is will depend upon how good our software model is.



Activity 9 Consider a computer system that will allow items to be reserved from a library. Imagine one such item that you may like to reserve and list two or three things that a librarian may want to know about this item.

Feedback 9 You may have thought of a book you wish to reserve in which case the librarian may need to know the title of the book and its author.

For every object we create in a system we need to define the attributes of that object i.e. the things we need to know about it.

Activity 10 Note that we can consider a reservation as an intellectual object (where the actual item is a physical object). Considering this intellectual object (item reservation) list two or three actions the librarian may need to perform on this object.

Feedback 10 The librarian may need to cancel this reservation (if you change your mind) they may also need to tell you if the item has arrived in stock for you to collect.

For each object we need to define the operations that will be performed on that object (as well as its attributes).

Activity 11 Considering the most general category of object that can be borrowed from a library, a 'loan item', list two or three more specific subcategory of object a library can lend out.

Feedback 11 Having defined the most general category of object (we call this a class) – something that can be borrowed – we may want to define more specialised sub-classes (e.g. books, magazines, audio/visual material). These will share the attributes defined for the general class but will have specific differences (for example there could be a charge for borrowing audio/visual items).

1.6 The Benefits of the Object Oriented Programming Approach

Whether or not you develop programs in an object oriented way, before you write the software you must first develop a model of what that software must be able to do and how it should work. Object oriented modelling is based on the ideas of abstraction, encapsulation, inheritance and polymorphism.

The general proponents of the object oriented approach claims that this model provides:

- better abstractions (modelling information and behaviour together)
- better maintainability (more comprehensible, less fragile software)
- better reusability (classes as encapsulated components that can be used in other systems)

We will look at these claims throughout this book and in Chapter 11 we will see a case study showing in detail how object oriented analysis works and how the resultant models can be implemented in an object oriented programming language (i.e. C#).

1.7 Software Implementation

Before a computer can complete useful tasks for us (e.g. check the spelling in our documents) software needs to be written and implemented on the machine it will run on. Software implementation involves the writing of program source code and preparation for execution of this on a particular machine. Of course before the software is written it needs to be designed and at some point it needs to be tested. There are many iterative lifecycles to support the process of design, implementation and testing that involve multiple implementation phases. Of particular concern here are the three long established approaches to getting source code to execute on a particular machine…

- compilation into machine-language object code
- direct execution of source code by 'interpreter' program
- compilation into intermediate object code which is then interpreted by run-time system

Implementing C# programs involves compiling the source code (C#) into machine-language object code. This approach has some advantages and disadvantages and it is worth comparing these three options in order to appreciate the implications for the C# developer.

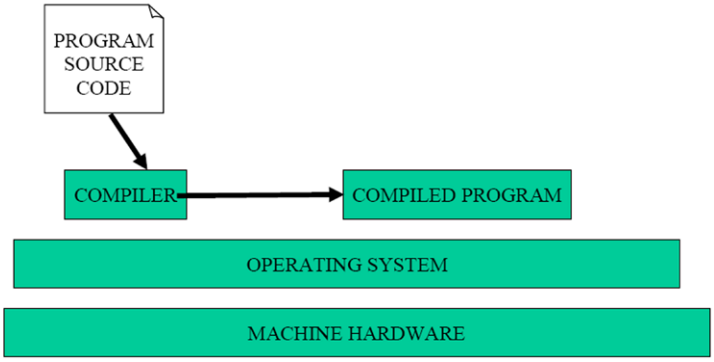
Compilation

The compiler translates the source code into machine code for the relevant hardware / operating system combination.

Strictly speaking there are two stages: compilation of program units (usually files), followed by 'linking' when the complete executable program is put together including the separate program units and relevant library code etc.

The compiled program then runs as a 'native' application for that platform.

This is the oldest model, used by early languages like Fortran and Cobol, and many modern ones like C#. It allows fast execution speeds but requires re-compilation of the program each time the code is changed or each time we want to run this code on a machine with a different operating system.



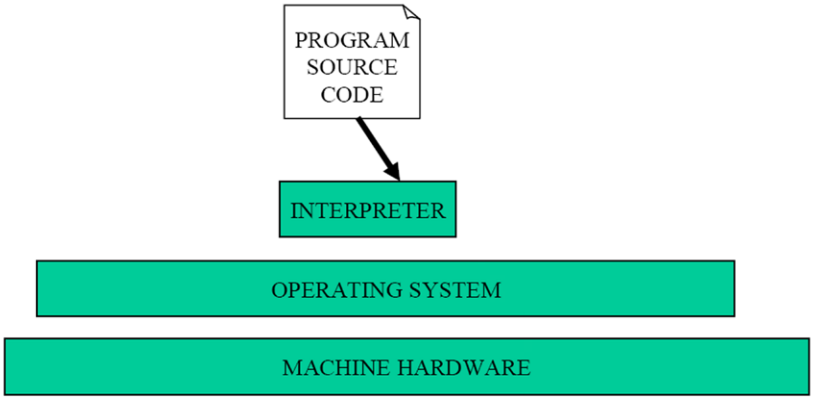
Interpretation

Here the source code is not translated into machine code. Instead an interpreter reads the source code and performs the actions it specifies.

We can say that the interpreter is like a 'virtual machine' whose machine language is the source code language.

No re-compilation is required after changing the code, but the interpretation process inflicts a significant impact on execution speed.

Scripting languages tend to work in this way.



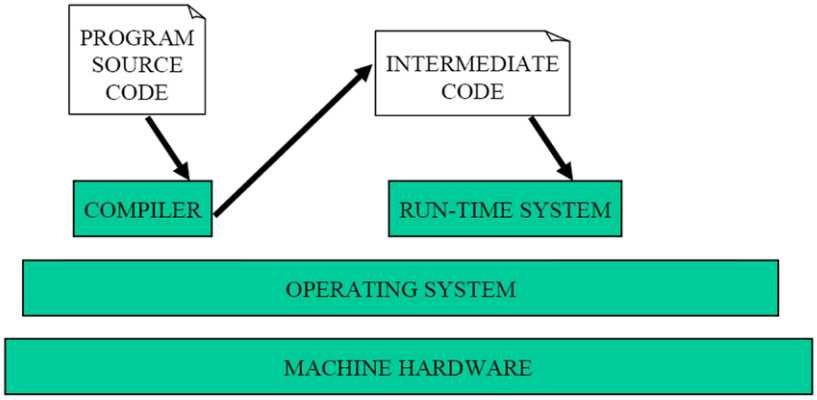
Interpreted programs can be slow but can work on any machine that has an appropriate interpreter. They do not need to be complied for different machines.

Intermediate Code

This model is a hybrid between the previous two.

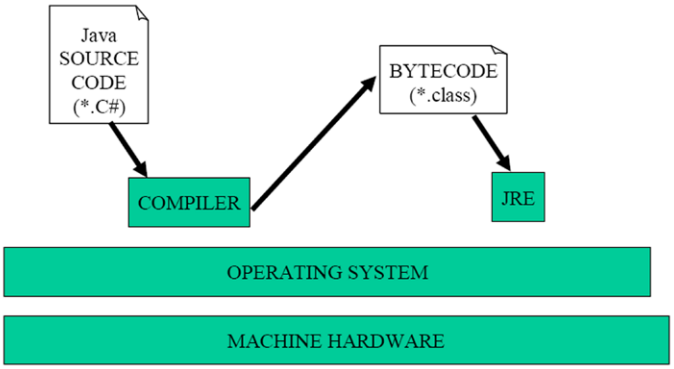
Compilation takes place to convert the source code into a more efficient intermediate representation which can be executed by a 'run-time system' (again a sort of 'virtual machine') more quickly that direct interpretation of the source code. However, the use of an intermediate code which is then executed by run-time system software allows the compilation process to be independent of the operating system / hardware platform, i.e. the same intermediate code should run on different platforms so long as an appropriate run-time system is available for each platform.

This approach is long-established (e.g. in Pascal from the early 1970s) and is how Java works. Java is a modern Object Oriented Language which is an alternative to C# and yet shares many similar features from the programmers point of view.



Running Java Programs

To run Java programs we must first generate intermediate code (called bytecode) using a compiler available as part of the Java Development Kit (JDK). Thus a Java compiler does not create .exe files i.e. code that could run directly on a specific machine but instead generates .class files.



A version of the Java Runtime Environment (JRE), which incorporates a Java Virtual machine (VM), is required to execute the bytecode and the Java library packages. Thus a JRE must be present on any machine which is to run Java programs.

The Java bytecode is standard and platform independent and as JRE's have been created for most computing devices (including PC's, laptops, mobile devices, mobile phones, internet devices etc) this makes Java programs highly portable and by compiling the code to an intermediate language Java strives to attain the fast implementation speeds obtained by fully compiled systems. i.e. Once complied Java code can run on any machine with a JRE irrespective of its underlying operating system without needing to be recompiled.

Running C# Programs

As we will see in section 1.8, C# programs, like Java programs, are also converted into an intermediate language but a C# compiler then goes further and links these with the necessary dll files to generate programs that can be directly executed on the target machine. C# programs are therefore fully complied programs i.e. .exe files are generated. While these are fast and efficient the source code must be recompiled each time we wish to run the program on a machine with a different operating system.

C# it is part of the .NET framework that aims to deliver additional benefits for the programmer.

Some programmers mistakenly believe that C# was written to create programmes only for the Windows operating system this is not true as we will see when we consider the .NET framework.

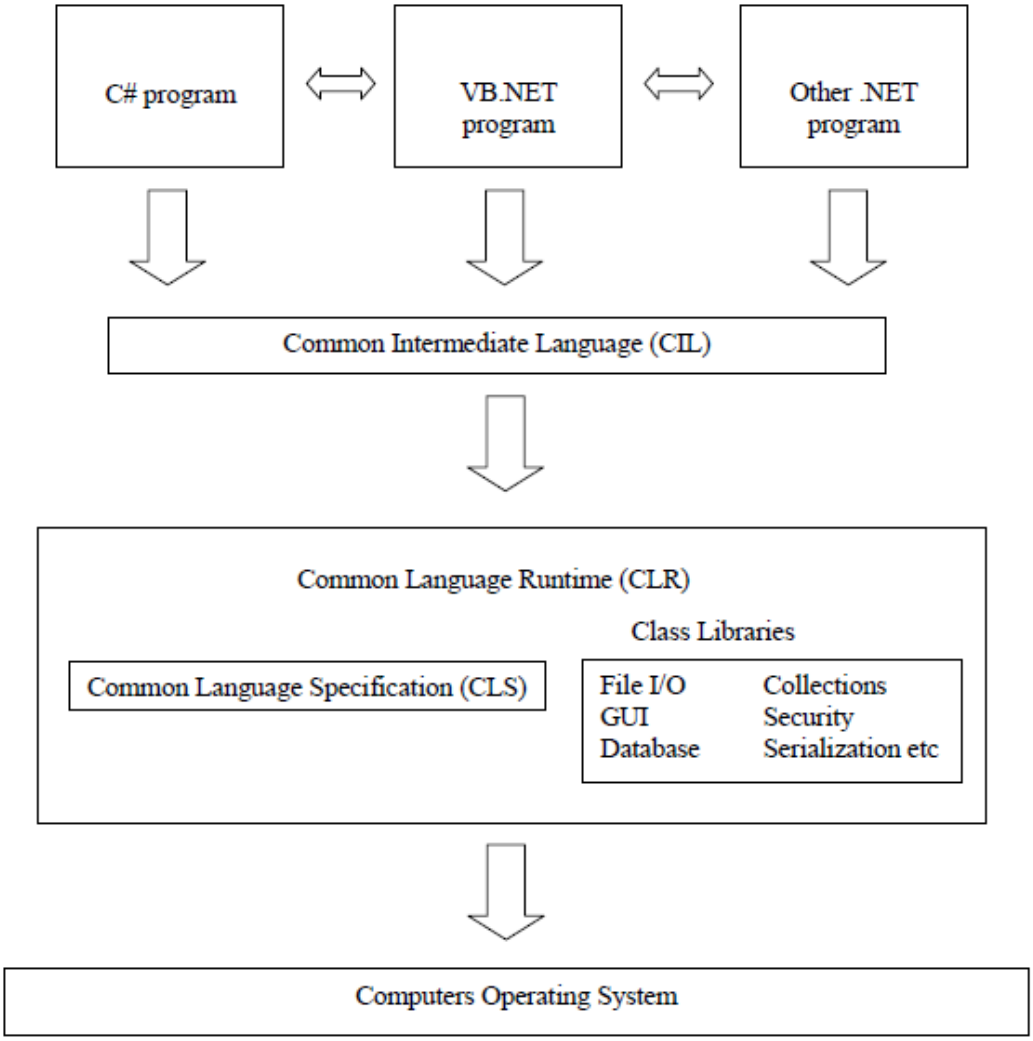
1.8 An Introduction to the .NET Framework

The .NET framework was designed to make life simpler for programmers by providing a common platform that can be used when writing programs in several different programming languages (C# and Visual Basic among others).

The .NET platform provides common features that can be used by programmers of all .NET languages. These include :-

a) a Common Language Runtime (CLR) engine that allows all .NET programs to run and use common features based on a Common Language Specification (CLS). The CLR also shields the programmer from having to deal with issues that are specific to individual operating systems.
b) a comprehensive class library providing a wealth of in built functionality (GUI, File I/I, Database, Threading, Serialization, Security, Web applications, Networking etc)
c) a Common Intermediate Language (CIL). All .NET source code is compiled into a common intermediate language this provides the ability to integrate code written in any .NET language making use of common exception handling facilities.

This relationship can be shown diagrammatically as below :-



C# programs are thus partly compiled into a common intermediate language. This is then linked with a CLR engine to give a .exe file.

Taken all together the .NET framework does not mean that all of the .NET languages are identical but they do have access to an identical library of methods that can be used by all .NET Languages. Thus learning to programme one .NET language makes it easier to learn and program another .NET language.

.NET programs can only run where a CLR engine has been created for the underlying operating system. Windows Vista and Windows 7 comes with a CLR engine and can therefore run .NET programs but older versions of windows may need a CLR engine installed before .NET programs can be run.

To enable .NET programs to be platform independent Microsoft published an open source specification for a CLR engine (called a Virtual Execution System) this included the definition of the C# language and a Common Language Infrastructure (CLI).

By using these definitions CLR engines have been created for other operating systems (e.g. Mac OS and Linux) and by using these .NET programs can run on different platforms… not just Microsoft Windows. However before programs can be run on these different platforms they do need to be recompiled for each platform thus .NET programs, including C# programs, are not as portable as Java programs.

While CLR engines are perhaps not as widely available as JREs they do exist for other platforms. One example of an open source CLR is Mono (www.mono-project.com). Mono is an open source, cross-platform, implementation of C# and the CLR that is compatible with Microsoft.NET. One part of the Mono project is MonoTouch. MonoTouch allows you to create C# and .NET apps for iPhone and iPod Touch, while taking advantage of iPhone APIs.

As well as compiling our C# programs it is possible to create a software installation routine that will download via the web a .NET runtime environment and automatically install this along with our software (assuming a .NET runtime environment is not already installed).

We will be writing and running code written in C# throughout this book and in doing so we will be making use of a compiler, the Common Language Runtime (CLR) engine and some of the more common class libraries. However the prime aim of this book is to teach generic Object Oriented programming and modelling principles that are common across a range of OO languages – not just .NET languages.

While we will illustrate OO principles in this book with C# code we will not concern ourselves further with the intricacies of how the CLR engine works, details regarding how .NET programs are compiled nor the detailed operation of the Common Intermediate Language (CIL).

However in order that the examples illustrated in this book can be demonstrated as practical worked examples we will introduce two modern Interactive Development Environments (IDEs), and some other tools specifically designed for the creation of .NET programs (see Chapter 8).

1.9 Summary

Object oriented programming involves the creation of classes by modelling the real world. This allows more specialised classes to be created that inherit the behaviour of the generalised classes. Polymorphic behaviour means that systems can be changed, as business needs change, by adding new specialised classes and these classes can be accessed by the rest of the system without any regard to their specialised behaviour and without changing other parts of the current system.

We will return to each of the concepts introduced here throughout the book and hopefully, by the end, you will have a good understanding of these concepts and understand how to apply them using C#.

2 The Unified Modelling Language (UML)

Introduction

This chapter will introduce you to the roles of the Unified Modelling Language (UML) and explain the purpose of four of the most common diagrams (class diagrams, object diagrams, sequence diagrams and package diagrams). These diagrams are used extensively when describing software designed according to the object oriented programming approach. Throughout this book particular emphasis will be placed on class diagrams as these are the most used part of the UML notation.

Objectives

By the end of this chapter you will be able to….

- Explain what UML is and explain the role of four of the most common diagrams,
- Draw class diagrams, object diagrams, sequence diagrams and package diagrams.

The material covered in this chapter will be expanded on throughout later chapters of the book and the skills developed here will be used in later exercises (particularly regarding class diagrams.

This chapter consists of six sections :-

1) An introduction to UML
2) UML Class Diagrams
3) UML Syntax
4) UML Package Diagrams
5) UML Object diagrams
6) UML Sequence Diagrams

2.1 An Introduction to UML

The Unified Modelling Language, UML, is sometimes described as though it was a methodology. It is not!

A methodology is a system of processes in order to achieve a particular outcome e.g. an organised sequence of activities in order to gather user requirements. UML does not describe the procedures a programmer should follow – hence it is not a methodology. It is, on the other hand, a precise diagramming notation that will allow program designs to be represented and discussed. As it is graphical in nature it becomes easy to visualise, understand and discuss the information presented in the diagram. However, as the diagrams represent technical information they must be precise and clear – in order for them to work - therefore there is a precise notation that must be followed.

As UML is not a methodology it is left to the user to follow whatever processes they deem appropriate in order to generate the designs described by the diagrams. UML does not constrain this – it merely allows those designs to be expressed in an easy to use, but precise, graphical notation.

A process will be explained in chapter 6 that will help you to generate good UML designs. Developing good designs is a skill that takes practise to this end the process is repeated in the case study (chapter 11). For now we will just concentrate on the UML notation not these processes.

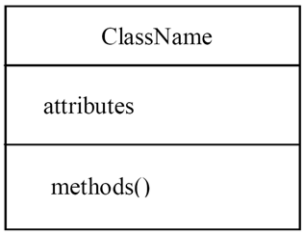
2.2 UML Class diagrams

Classes are the basic components of any object oriented software system and UML class diagrams provide an easy way to represent these. As well as showing individual classes, in detail, class diagrams show multiple classes and how they are related to each other. Thus a class diagram shows the architecture of a system.

A class consists of :-

- a unique name (conventionally starting with an uppercase letter)
- a list of attributes (int, double, boolean, String etc)
- a list of methods

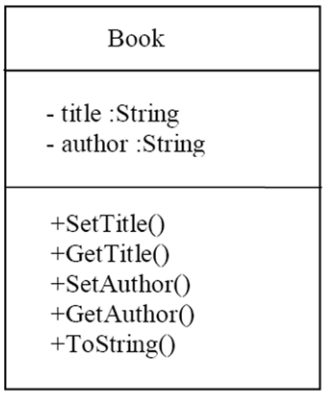
This is shown in a simple box structure…



For attributes and methods visibility modifiers are shown (+ for public access, – for private access). Attributes are normally kept private and methods are normally made public.

Accessor methods are created to provide access to private attributes when required. Thus a public method SetTitle() can be created to change the value of a private attribute 'title'.

Thus a class Book, with String attributes of title and author, and the following methods SetTitle(), GetTitle(), SetAuthor(), GetAuthor() and ToString() would be shown as ….



Note: String shown above is not a primitive data type but is itself a class. Hence it starts with a capital letter.

A Note On Naming Conventions

Some programmers use words beginning in capitals to denote class names and words beginning in lowercase to represent attributes or methods (thus ToString() would be shown as toString()). This is a common convention when designing and writing programs in Java (another common OO language). However it is not a convention followed by C# programmers – where method names usually start in Uppercase. Method names can be distinguished from class names by the use of (). This in the example above.

'Book' is a class
'title' is an attribute and
'SetTitle()' is a method.

UML diagrams are not language specific thus a software design, communicated via UML diagrams, can be implemented in a range of OO languages.

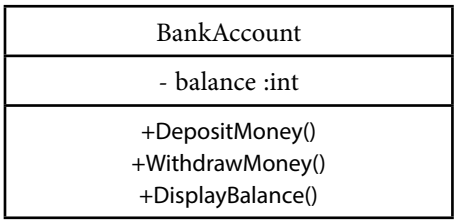
Furthermore traditional accessor methods, getters and setters, are not required in C# programs as they are replaced by 'properties'. Properties are in effect hidden accessor methods thus the getter and setter methods shown above, GetTitle(), SetTitle() etc are not required in a C# program. In C# an attribute would be defined called 'title' and a property would be defined as 'Title'. This would allow us to set the 'title' directly by using the associated property 'Title =…..;'.

The UML diagrams shown in this book will use the naming convention common among C# programmers … for the simple reason that we will be writing sample code in C# to demonstrate the OO principles discussed here. Though initially we will show conventional assessor methods these will be replaced with properties when coding.

Activity 1

Draw a diagram to represent a class called 'BankAccount' with the attribute balance (of type int) and methods DepositMoney(), WithdrawMoney() and DisplayBalance(). Show appropriate visibility modifiers.

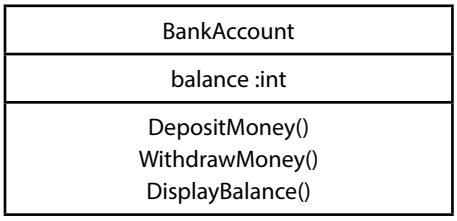
Feedback 1



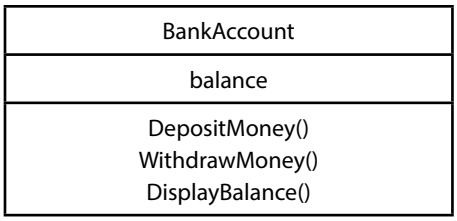
The diagram above shows this information

UML allows us to suppress any information we do not wish to highlight in our diagrams – this allows us to suppress irrelevant detail and bring to the readers attention just the information we wish to focus on. Therefore the following are all valid class diagrams…

Firstly with the access modifiers not shown….



Secondly with the access modifiers and the data types not shown…..



And finally with the attributes and methods not shown…..

i.e. there is a class called 'BankAccount' but the details of this are not being shown.

Of course virtually all C# programs will be made up of many classes and classes will relate to each other – some classes will make use of other classes. These relationships are shown by arrows. Different type of arrow indicate different relationships (including inheritance and aggregation relationships).

In addition to this class diagrams can make use of keywords, notes and comments.

As we will see in examples that follow, a class diagram can show the following information :-

- Classes
  - attributes
  - operations
  - visibility

- Relationships
    - navigability
    - multiplicity
    - dependency
    - aggregation
    - composition
- Generalization / specialization
    - inheritance
    - interfaces
- Keywords
- Notes and Comments

2.3 UML Syntax

As UML diagrams convey precise information there is a precise syntax that should be followed.

Attributes should be shown as: *visibility name* : *type multiplicity*

Where visibility is one of :-

- '+' public
- '-' private
- '#' protected
- '~' package

and Multiplicity is one of :-

- 'n' exactly n
- '*' zero or more
- 'm..'n' between m and n

The following are examples of attributes correctly specified using UML :-

- custRef : int [1]

a private attribute custRef is a single int value
this would often be shown as - **custRef : int** However with no multiplicity shown we cannot safely assume a multiplicity of one was intended by the author.

 # itemCodes : String [1..*]

a protected attribute itemCodes is one or more String values

validCard : boolean

an attribute validCard, of unspecified visibility, has unspecified multiplicity

Operations also have a precise syntax and should be shown as:

visibility name (par1 : type1, par2 : type2): returntype

where each parameter is shown (in parenthesis) and then the return type is specified.

An example would be

+ AddName (newName : String) : boolean

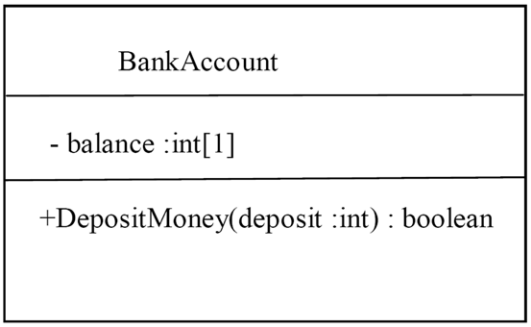
This denotes a public method 'AddName' which takes one parameter 'newName' of type String and returns a boolean value.

Activity 2

Draw a diagram to represent a class called 'BankAccount' with a private attribute balance (this being a single integer) and a public method DepositMoney() which takes an integer parameter, 'deposit' and returns a boolean value. Fully specify all of this information on a UML class diagram.

Feedback 2

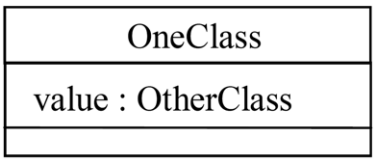
The diagram below shows this information



Denoting Relationships

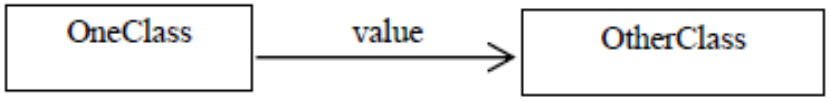
As well as denoting individual classes, Class diagrams denote relationships between classes. One such relationships is called an 'Association'. We will learn a lot more about associations in Chapter 6 where we look at how to model an object oriented system. Here we are just learning the UML notation we will be using later.

In a class attributes will be defined. These could be primitive data types (int, boolean etc.) however attributes can also be complex objects as defined by other classes.



Thus the figure above shows a class 'OneClass' that has an attribute 'value'. This value is not a primitive data type but is an object of type defined by 'OtherClass'.

We could denote exactly the same information by the diagram below.

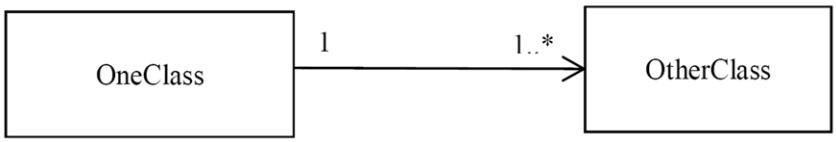


We use an association when we want to give two related classes, and their relationship, prominence on a class diagram

The 'source' class points to the 'target' class.

Strictly we could use an association when a class we define has a String instance variable – but we would not do this because the String class is part of the C# platform and 'taken for granted' like an attribute of a primitive type. This would generally be true of all library classes unless we are drawing the diagram specifically to explain some aspect of the library class for the benefit of someone unfamiliar with its purpose and functionality.

Additionally we can show multiplicity at both ends of an association:



This implies that 'OneClass' maintains a collection of objects of type 'OtherClass'. Collections are an important part of the C# library that we will look at the use of collections in Chapter 7.

Activity 3

Draw a diagram to represent a class called 'Catalogue' and a class called 'ItemForSale' as defined below :-

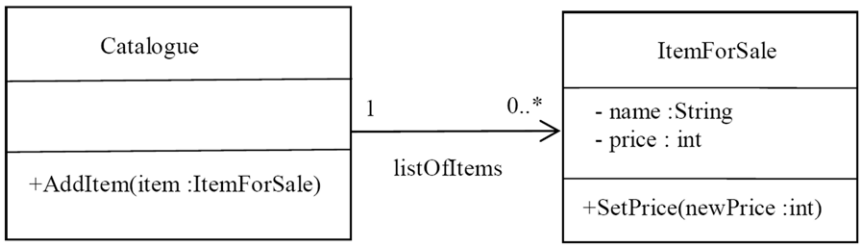
ItemForSale has an attribute 'name' of type String and an attribute 'price' of type int. It also has a method SetPrice() which takes an integer parameter 'newPrice'.

'Catalogue' has an attribute 'listOfItems' i.e. the items currently held in the catalogue. As zero or more items can be stored in the catalogue 'listOfItems' will need to be an array or collection. 'Catalogue' also has one method AddItem() which takes an 'item' as a parameter (of type ItemForSale) and adds this item to the 'listOfItems'.

Draw this on a class diagram showing appropriate visibility modifiers for attributes and methods.

Feedback 3

The diagram below shows this information

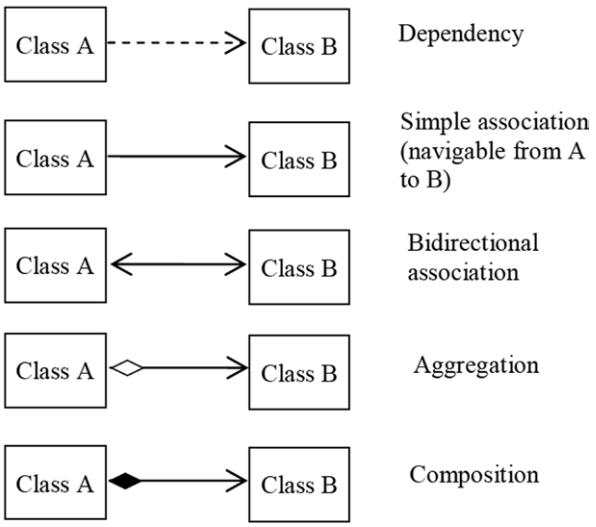


Note: According to the naming convention followed here all class names begin in uppercase, attribute names begin in lowercase, method names begin in uppercase and use () to distinguish them from class names. Also note that the class ItemForSale describes a single item (not multiple items). 'listOfItems' however maintains a list of zero or more individual objects.

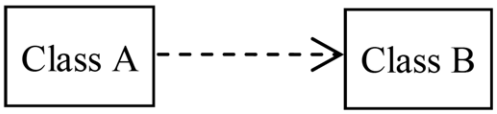
Types of Association

There are various different types of association denoted by different arrows:-

- Dependency,
- Simple association
- Bidirectional association
- Aggregation and
- Composition



Dependency

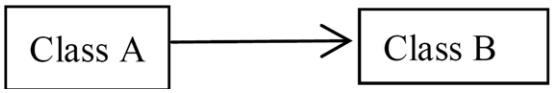


- Dependency is the most unspecific relationship between classes (not strictly an „association')
- Class A in some way uses facilities defined by Class B
- Changes to Class B may affect Class A

Typical use of dependency lines would be where Class A has a method which is passed a parameter object of Class B, or uses a local variable of that class, or calls 'static' methods in Class B.

Example: A Print() method may require a printer object as a parameter. Each time the Print() method is invoked a different printer object could be passed as a parameter and thus the printout will appear in a different place. Thus while the class containing the Print() method requires a printer object to work it does not need to be permanently associated with one specific printer.

Simple Association

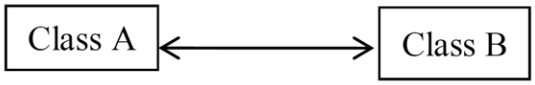


- In an association Class A 'uses' objects of Class B
- Typically Class A has an attribute of Class B
- Navigability is from A to B:
  i.e. A Class A object can access the Class B object(s) with which it is associated. The reverse is not true – the Class B object doesn't 'know about' the Class A object

A simple association typically corresponds to an instance variable in Class A of the target class B type.

Example: the **Catalogue** above needs access to 0 or more ItemForSale so items can be added or removed from a Catalogue. An **ItemForSale** does not need to access a **Catalogue** in order to set its price or perform some other method associated with the item itself.

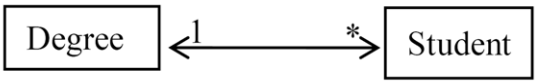
Bidirectional Association



- Bidirectional Association is when Classes A and B have a two-way association
- Each refers to the other class
- Navigability A to B and B to A:
  -A Class A object can access the Class B object(s) with which it is associated
  -Object(s) of Class B 'belong to' Class A
  -Implies reference from A to B
  -Also, a Class B object can access the Class A object(s) with which it is associated

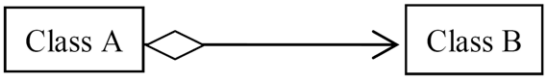
A bidirectional association is complicated because each object must have a reference to the other object(s) and generally bidirectional associations are much less common than unidirectional ones.

An example of a bidirectional association may between a 'Degree' and 'Student'. ie. given a Degree we may wish to know which Students are studying on that Degree. Alternatively starting with a student we may wish to know the Degree they are studying.



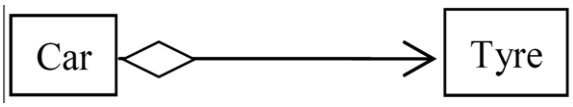
As many students study the same Degree at the same time, but students usually only study one Degree there is still a one to many relationship here (of course we could model a situation where we record degrees being studied and previous degrees passed – in this case, as a student may have passed more than one degree, we would have a many to many relationship).

Aggregation



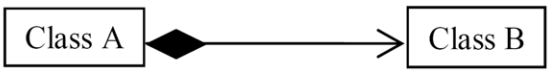
- Aggregation denotes a situation where Object(s) of Class B 'belong to' Class A
- Implies reference from A to B
- While aggregation implies that objects of Class B belong to objects of Class A it also implies that object of Class B retain an existence independent of Class A. Some designers believe there is no real distinction between aggregation and simple association

An example of aggregation would be between a class Car and a class Tyre



We think of the tyres as belonging to the car they are on, but at the garage they may be removed and placed on a rack to be repaired. Their existence isn't dependent on the existence of a car with which they are associated. Some designers believe that aggregation can be replaced as a simple association as when implementing this design in a program it makes no difference to the programmer.

Composition



- Composition is similar to aggregation but implies a much stronger belonging relationship i.e. Object(s) of Class B are 'part of' a Class A object
- Again implies reference from A to B
- Much 'stronger' than aggregation in this case Class B objects are an integral part of Class A and in general objects of Class B never exist other than as part of Class A, i.e. they have the same 'lifetime'

An example of composition would be between Points, Lines and Shapes as elements of a Picture. These objects can only exist as part of a picture, and if the picture is deleted they are also deleted.

As well as denoting associations, class diagrams can denote :-

- Inheritance,
- Interfaces,
- Keywords and
- Notes

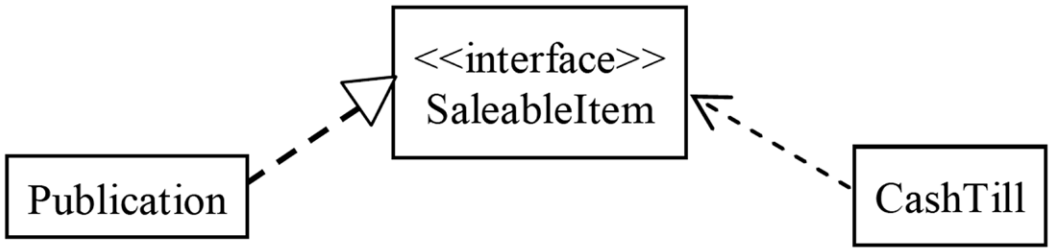
Inheritance

- Aside from associations, the other main modelling relationship is inheritance:
- Class A 'inherits' both the interface and implementation of Class B, though it may override implementation details and supplement both.

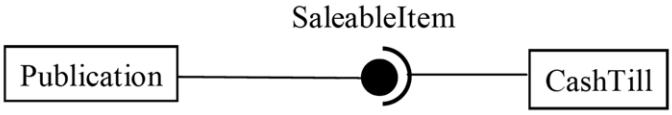
We will look at inheritance in detail in Chapter 3.

Interfaces

- Interfaces are similar to inheritance however with interfaces only the interface is inherited. The methods defined by the interface must be implemented in every class that implements the interface.
- Interfaces can be represented using the <<interface>> keyword:



There is also a shorthand for this



In both cases these examples denote that the SaleableItem interface is **required by** CashTill and **implemented by** Publication.

NB the dotted-line version of the inheritance line/arrow which shows that Publication 'implements' or 'realizes' the SaleableItem interface.

The "ball and socket" notation is new in UML 2 – it is a good visual way of representing how interfaces connect classes together.

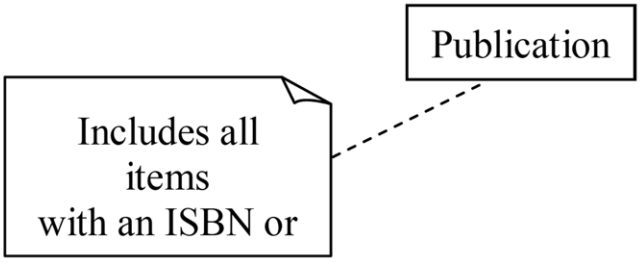
We will look at the application of interfaces in more detail in Chapter 4.

Keywords

UML defines keywords to refine the meaning of the graphical symbols
We have seen <<interface>> and we will also make use of <<abstract>> but there are many more.

An abstract class may alternatively be denoted by showing its name in *italics* though this is perhaps less obvious to a casual reader.

Notes



Finally we can add notes to comment on a diagram element. This gives us a 'catch all' facility for adding information not conveyed by the graphical notation.

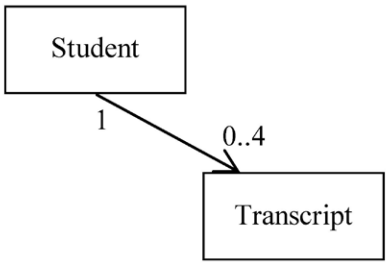
Activity 4

From your own experience, try to develop a model which illustrates the use of the following elements of UML Class Diagram notation:

- simple association
- bidirectional association
- aggregation (tricky!)
- composition
- association multiplicity
- generalization (inheritance)
- interfaces
- notes

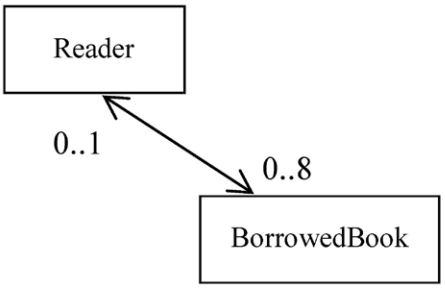
For this exercise concentrate on the relationships between classes rather than the details of their members. If possible explain and discuss your model with other students or friends.

To help you get started some small examples are given below:-



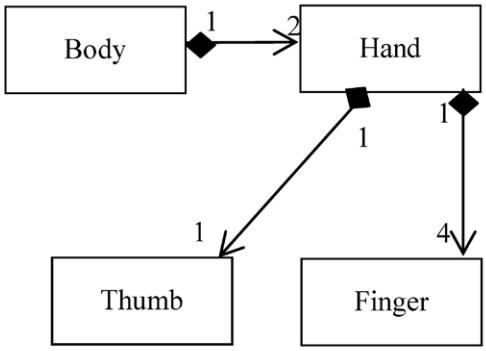
In a University administration system we might produce a transcript of results for each year the student has studied (including a possible placement year).

This association relationship is naturally unidirectional – given a student we might want to find their transcript(s), but it seems unlikely that we would have a transcript and need to find the student to whom it belonged.



In a library a reader can borrow up to eight books. A particular book can be borrowed by at most one reader.

We might want a bidirectional relationship as shown here because, in addition to being able to identify all the books which a particular reader has borrowed, we might want to find the reader who has borrowed a particular book (for example to recall it in the event of a reservation).

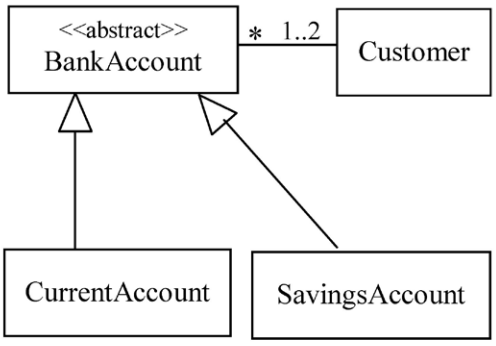


This might be part of the model for some kind of educational virtual anatomy program.

Composition – the "strong" relationship which shows that one object is (and has to be) part of another seems appropriate here.

The multiplicities would not always work for real people though – they might have lost a finger due to accident or disease, or have an extra one because of a genetic anomaly.

But what if we were modelling the "materials" in a medical school anatomy lab? A hand might not always be part of a body! Perhaps the "weaker" aggregation relationship would reflect this better.

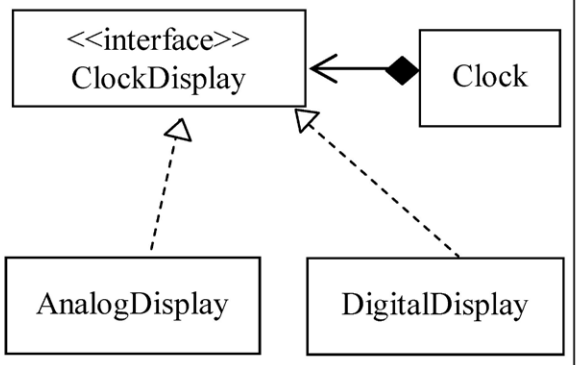


A customer can have any number of bank accounts, and a bank account can be held by one person or two people (a "joint account"). We have suppressed the navigability of this relationship, perhaps because we have not yet decided this issue.

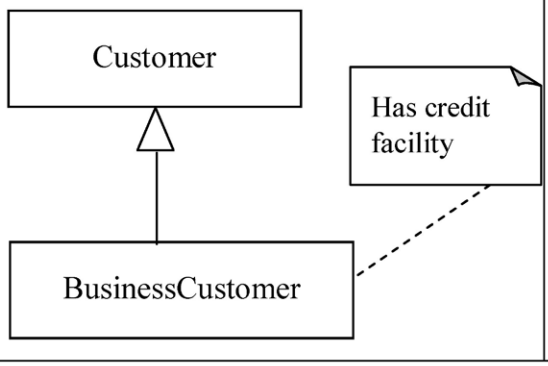
A bank account **must** either be a current account or a savings account – hence BankAccount itself is abstract.

(We could have shown this using italics rather than the <<abstract>> keyword)

BankAccount



Part of a clock is a display to show the time. This might be an analogue display or a digital display. We could use a superclass and two subclasses, but since the implementation of the two displays will be entirely different it may be more appropriate to use an interface to define the operations which AnalogDisplay and DigitalDisplay must provide.



A note is used here to add some information which cannot be conveyed with standard UML notation and may not be obvious simply from the names and relationships of the classes depicted.

Feedback 4

There is no specific feedback for this activity.

2.4 UML Package Diagrams

While class diagrams are the most commonly used diagram of those defined in UML notation, and we will make significant use of these throughout this book, there are other diagrams that denote different types of information. Here we will touch upon three of these :-

- Package Diagrams
- Object Diagrams and
- Sequence Diagrams

World maps, country maps and city maps all show spatial information, just on different scales and with differing levels of detail. Large OO systems can be made up of hundreds, or potentially thousands, of classes and thus if the class diagram was the only way to represent the architecture of a large system it would become overly large and complex. Thus, just as we need world maps, we need package diagrams to show the general architecture of a large system. Even modest systems can be broken down into a few basic components i.e. packages. We will see an example of packages in use in Chapter 11. For now we will just look at the package diagramming notation.

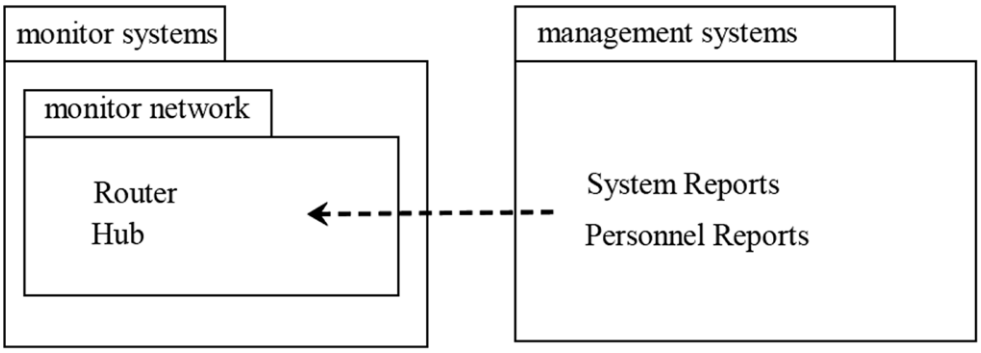
Packages diagrams allow us to provide a level of organisation and encapsulation above that of individual classes Packages are implemented in C'# by creating subfolders and defining a 'namespace'. When writing a large system in C# we use this to segment a large system into smaller more manageable sub-systems. We denote these sub-systems using package diagrams during the design stage.

A large C# development should be split into suitable packages at the design stage
UML provides a 'Package Diagram' to represent the relationships between classes and packages.

We can depict

- classes within packages
- nesting of packages
- dependencies between packages

In the diagram below we see two packages :- 'monitor systems' and 'management systems' These depict part of a large system for a multinational corporation to manage and maintain their operations including their computer systems and personnel.



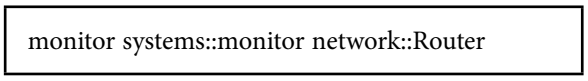
Looking at this more closely we can see that inside the 'monitor systems' package is another called 'monitor network'. This package contains at least two classes 'Router' and 'Hub' though presumably it contains many other related classes.

The package 'management systems' contains two classes (or more) 'System Reports' and 'Personnel Reports'.

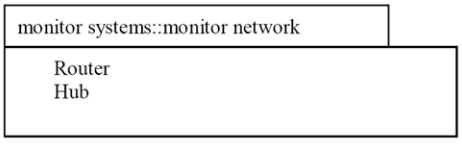
Furthermore we can see that the classes inside the package 'management systems' in some way use the classes inside 'monitor network'. Presumably it is the 'System Reports' class that makes use of these as it needs to know about the status of the network.

Note that the normal UML principle of suppression applies here – these packages may contain other packages and each package may contain dozens of classes but we simply choose not to show them.

In the class diagram below we have an alternative way of indicating that 'Router' is a class inside the 'monitor network' package, which is in turn inside 'monitor systems' package.



And again below a form which shows both classes more concisely than at the top.



These different representations will be useful in different circumstances depending on what a package diagram is aiming to convey.

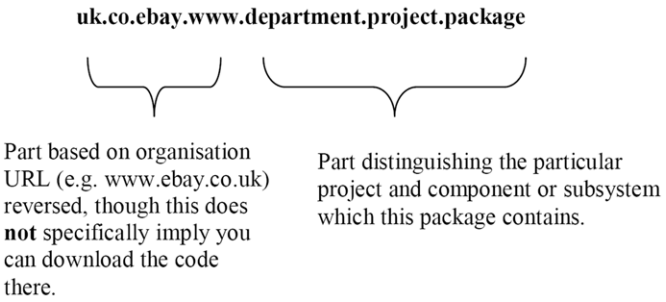
Package Naming

By convention, package names are normally in lowercase. We will follow this convention we will as it helps to distinguish between packages and classes.

For local individual projects packages could be named according to personal preference, e.g.

mysystem
mysystem.interface
mysystem.engine
mysystem.engine.util
mysystem.database

However, packages are often distributed and to enable this packages need globally unique names, thus a naming convention has been adopted based on URLs



Note on a package diagram each element is not separated by a '.' but by '::'.

Activity 5

You and a flatmate decide to go shopping together. For speed split the following shopping list into two halves – items to be collected by you and items to be collected by your flatmate.

Apples, Furniture polish, Pears, Carrots, Toilet Rolls, Potatoes, Floor cleaner. Matches, Grapes

Feedback 5

To make your shopping efficient you probably organised your list into two lists of items that are located in the same parts of the shop:-

List 1	List 2
Apples,	Furniture polish,
Pears,	Floor cleaner
Grapes	Matches
Carrots,	Toilet Rolls,
Potatoes	

Activity 6

You run a team of three programmers and are required to write a program in C# to monitor and control a network

system. The system will be made up of seven classes as described below. Organise these classes into three packages. Each programmer will then be responsible for the code in one package. Give the packages any name you feel appropriate.

Main	this class starts the system
Monitor	this class monitors the network for performance and breaches in security
Interface	this is a visual interface for entire system
Reconfigure	this allows the network to be reconfigured
RecordStats	this stores data regarding the network in a database
RemoteControlthis	allows some remote control over the system via telephone
PrintReports	this uses the data stored in the database to print management reports for the organisations management.

Feedback 6

When organising a project into packages there is not always 'one correct answer' but if you organise your classes into appropriate packages (with classes that have related functionality) you improve the encapsulation of the system and improve the efficiency of your programmers. A suggested solution to activity 6 is given below.

interface
- Main
- Interface
- RemoteControl

network
- Monitor
- Reconfigure

database
- RecordStats
- PrintReports

Activity 7

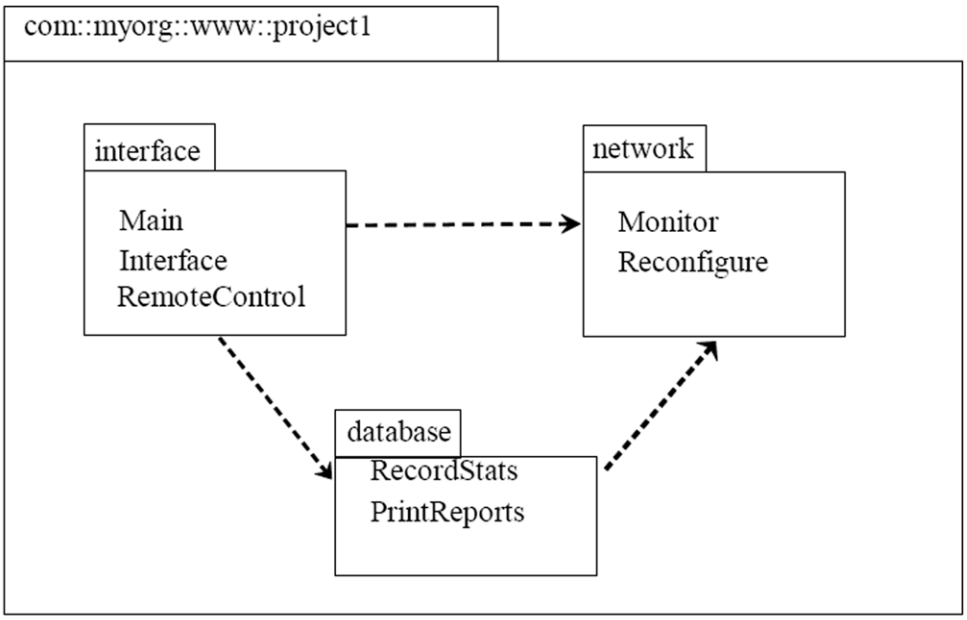
Assume the URL of your organisation is 'www.myorg.com' and the three packages and seven classes shown below are all part of 'project1'. Draw a package diagram to convey this information.

interface
- Main
- Interface
- RemoteControl

network
- Monitor
- Reconfigure

database
- RecordStats
- PrintReports

Feedback 7



Note: Dependency arrows have been drawn to highlight relationships between packages. When more thought has been put into determining these relationships they may turn out to be associations (a much stronger relationship than a mere dependency).

2.5 UML Object Diagrams

Class diagrams and package diagrams allow us to visualise and discuss the architecture of a system, however at times we wish to discuss the data a system processes. Object diagrams allow us to visual one instance of time and the data that a system may contain in that moment.

Object diagrams look superficially similar to class diagrams however the boxes represent specific instances of objects.

Boxes are titled with :-

objectName : ClassName

As each box describes a particular object at a specific moment in time the box contains attributes and their values (at that moment in time).

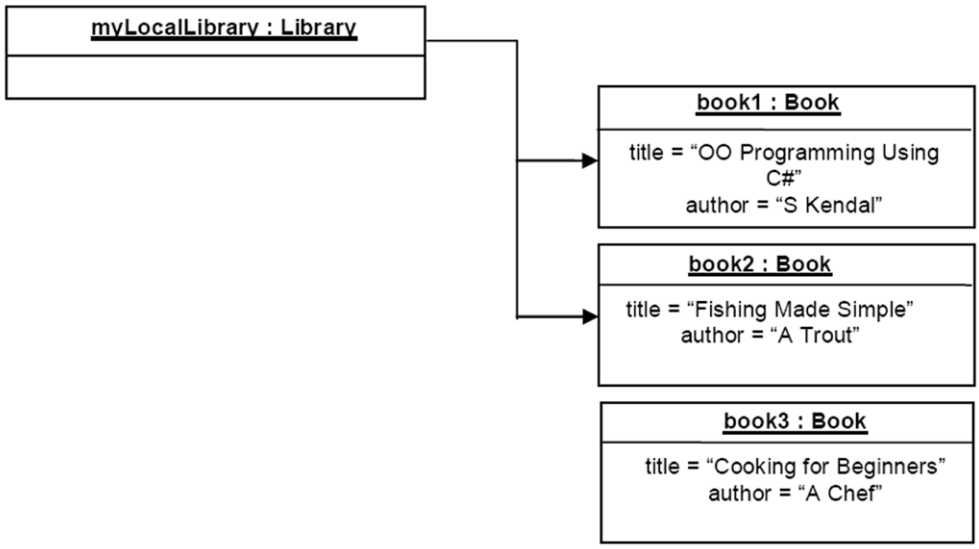
attribute = value

These diagrams are useful for illustrating particular 'snapshot' scenarios during design.

The object diagram below shows several object that may exist at a moment in time for a library catalogue system. The system contains two classes :-

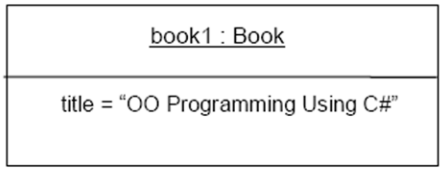
Book, which store the details of a book and

Library, which maintains a collection of books, with books being added, searched for or removed as required.

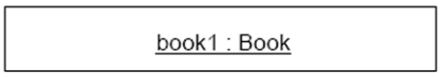


Looking at this diagram we can see that at a particular moment in time, while three books have been created only two have been added to the library. Thus if we were to search the library for 'Cooking for Beginners' we would not expect the book to be found.

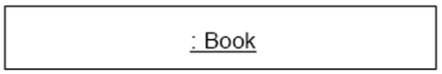
As with class diagrams, elements can be freely suppressed on object diagrams.
For example, all of these are legal:



- suppress some attributes



- suppress all attributes

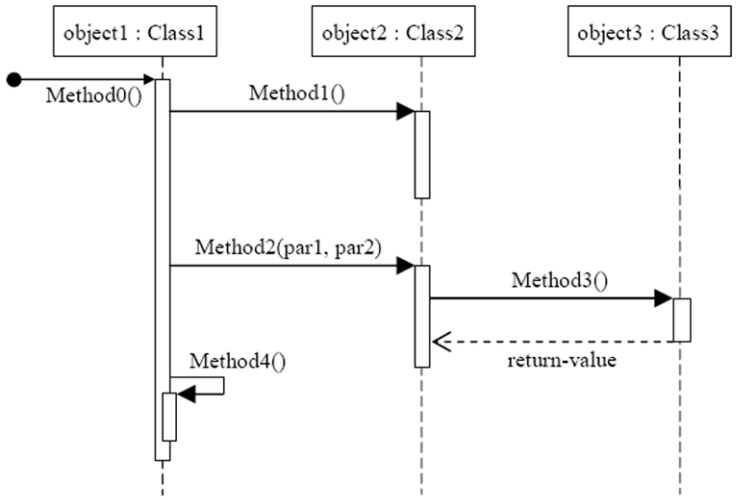


- suppress object name

2.6 UML Sequence Diagrams

Sequence diagrams are entirely different from class diagrams or object diagrams. Class diagrams describe the architecture of a system and object diagrams describe the state of a system at one moment in time. However sequence diagrams describe how the system works over a period of time. Sequence diagrams are 'dynamic' rather than 'static' representations of the system. They show the sequence of method invocations within and between objects over a period of time. They are useful for understanding how objects collaborate in a particular scenario.

See the example below :-



We have three objects in this scenario. Time runs from top to bottom, and the vertical dashed lines (lifelines) indicate the objects' continued existence through time.

This diagram shows the following actions taking place :-

- Firstly a method call (often referred to in OO terminology as a message) to Method0() comes to object1 from somewhere – this could be another class outside the diagram.
- object1 begins executing its Method0() (as indicated by the vertical bar (called an activation bar) which starts at this point.
- object1.Method0() invokes object2.Method1() – the activation bar indicates that this executes for a period then returns control to Method0()
- Subsequently object1.Method0() invokes object2.Method2() passing two parameters
- Method2() subsequently invokes object3.Method3(). When Method3() ends it passes a return value back to Method2()
- Method2() completes and returns control to object1.Method0()
- Finally Method0() calls another method of the same object, Method4()

Selection and Iteration

The logic of a scenario often depends on selection ('if') and iteration (loops).
There is a notation ('interaction frames') which allow ifs and loops to be represented in sequence diagrams however these tend to make the diagrams cluttered.
Sequence diagrams are generally best used for illustrating particular cases, with the full refinement reserved for the implementation code.

Fowler ("UML Distilled", 3rd Edn.) gives a brief treatment of these constructs.

2.7 Summary

UML is not a methodology but a precise diagramming notation.

Class diagrams and package diagrams are good for describing the architecture of a system. Object diagrams describe the data within an application at one moment in time and sequence diagrams describe how a system works over a period of time.

UML gives different meaning to different arrows therefore one must be careful to use the notation precisely as specified.

With any UML diagram suppression is encouraged – thus the author of a diagram can suppress any details they wish in order to convey essential information to the reader.

3 Inheritance and Method Overriding

Introduction

This chapter will discuss the essential concepts of Inheritance, method overriding and the appropriate use of the keyword 'Base'.

Objectives

By the end of this chapter you will be able to

- Appreciate the importance of an Inheritance hierarchy,
- Understand how to use Abstract classes to factor out common characteristics
- Override methods (including those in the 'Object' class),
- Explain how to use 'Base' to invoke methods that are in the process of being overridden,
- Document an inheritance hierarchy using UML and
- Implement inheritance and method overriding in C# programs.

All of the material covered in this chapter will be developed and expanded on in later chapters of this book. While this chapter will focus on understanding the application and documentation of an inheritance hierarchy, Chapter 6 will focus on developing the analytical skills required to define your own inheritance hierarchies.

This chapter consists of twelve sections :-

1) Object Families
2) Generalisation and Specialisation
3) Inheritance
4) Implementing Inheritance in C#
5) Constructors
6) Constructor Rules
7) Access Control
8) Abstract Classes
9) Overriding Methods
10) The 'Object' Class
11) Overriding ToString() defined in 'Object'
12) Summary

3.1 Object Families

Many kinds of things in the world fall into related groups of 'families'. 'Inheritance' is the idea 'passing down' characteristics from parent to child, and plays an important part in Object Oriented design and programming.

While you are probably already familiar with constructors, and access control (public/private), there are particular issues in relating these to inheritance we need to consider.

Additionally we need to consider the use of Abstract classes and method overriding as these are important concepts in the context of inheritance.

Finally we will look at the 'Object' class which has a special role in relation to all other classes in C#.

3.2 Generalisation and Specialisation

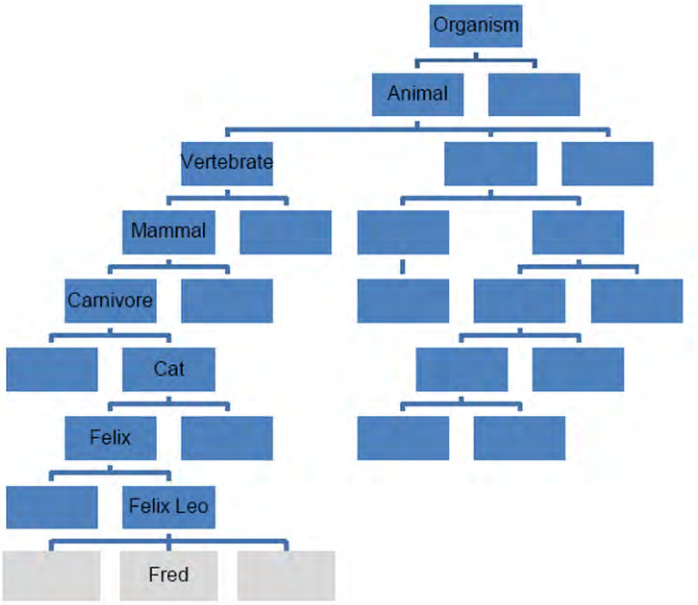
Classes are a generalized form from which objects with differing details can be created. Objects are thus 'instances' of their class. For example Student 051234567 is an instance of class Student. More concisely, 051234567 **is a** Student. Constructors are special methods that create an object from the class definition.

Classes themselves can often be organised by a similar kind of relationship.

One hierarchy, that we all have some familiarity with, is that which describes the animal kingdom :-

- Kingdom (e.g. animals)
- Phylum (e.g. vertebrates)
- Class (e.g. mammal)
- Order (e.g. carnivore)
- Family (e.g. cat)
- Genus (e.g. felix)
- Species (e.g. felix leo)

We can represent this hierarchy graphically ….



Of course to draw the complete diagram would take more time and space than we have available.

Here we can see one specific animal shown here :-'Fred'. Fred is not a class of animal but an actual animal.

Fred **is a** felix leo **is a** felix is a cat is a carnivore

Carnivores eat meat so Fred has the characteristic 'eats meat'.

Fred **is a** felix leo **is a** felix is a cat **is a** carnivore **is a** mammal **is a** vertebrate

Vertebrates have a backbone so Fred has the characteristic 'has a backbone'.

The 'is a' relationship links an individual to a hierarchy of characteristics. This sort of relationship applies to many real world entities, e.g. BonusSuperSaver **is a** SavingsAccount **is a** BankAccount.

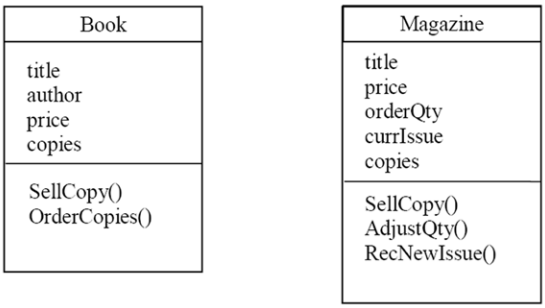
3.3 Inheritance

We specify the general characteristics high up in the hierarchy and more specific characteristics lower down. An important principle in OO – we call this **generalization** and **specialization**.

All the characteristics from classes above in a class/object in the hierarchy are automatically featured in it – we call this **inheritance.**

Consider books and magazines - both are specific types of publication.

We can show classes to represent these on a UML class diagram. In doing so we can see some of the instance variables and methods these classes may have.



Attributes 'title', 'author' and 'price' are obvious. Less obvious is 'copies' this is how many are currently in stock.

For books, OrderCopies() takes a parameter specifying how many extra copies are added to stock.

For magazines, orderQty is the number of copies received of each new issue and currIssue is the date/period of the current issue (e.g. "January 2011", "Fri 6 Jan", "Spring 2011" etc.) When a new issue is received the old issues are discarded and orderQty copies are placed in stock. Therefore RecNewIssue() sets currIssue to the date of new issue and restores copies to orderQty. AdjustQty() modifies orderQty to alter how many copies of subsequent issues will be stocked.

Activity 1

Look at the 'Book' and 'Magazine' classes defined above and identify the commonalities and differences between two classes.

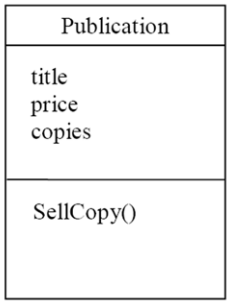
Feedback 1

These classes have three instance variables in common: title, price, copies.
They also have in common the method SellCopy().

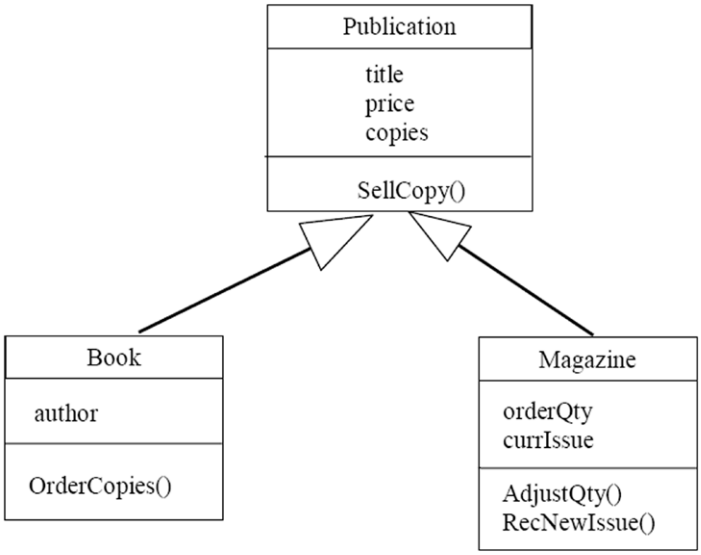
The differences are as follows…

Book additionally has author, and OrderCopies().
Magazine additionally has orderQty, currIssue, AdjustQty() and RecNewIssue().

We can separate out ('factor out') these common members of the classes into a superclass called Publication. In C# a superclass is often called a base class.



The differences will need to be specified as additional members for the 'subclasses' Book and Magazine.



In this is a UML Class Diagram the hollow-centred arrow denotes inheritance.

Note the subclass has the generalized superclass (or base class) characteristics + additional specialized characteristics. Thus the Book class has four instance variables (title, price, copies and author) it also has two methods (SellCopy() and OrderCopies()).

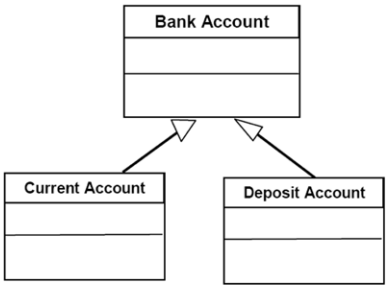
The inherited characteristics are **not** listed in subclasses. The arrow shows they are acquired from the superclass.

Activity 2

Arrange the following classes into a suitable hierarchy and draw these on a class diagram…

a current account
a deposit account
a bank account
Simon's deposit account

Feedback 2



The most general class goes at the top of the inheritance hierarchy with the other classes then inheriting the attributes and methods of this class.

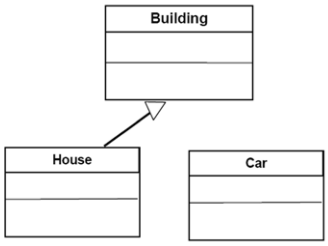
Simon's deposit account should not be shown on a class diagram as this is a specific instance of a class i.e. it is an object.

Activity 3

Arrange the following classes into a suitable hierarchy and draw these on a class diagram…

a building
a house
a car

Feedback 3



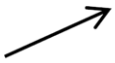
A house is a type of building and can therefore inherit the attributes of building however this is not true of a car. We cannot place two classes in an inheritance hierarchy unless we can use the term **is a.**

Note class names, as always, begin in uppercase.

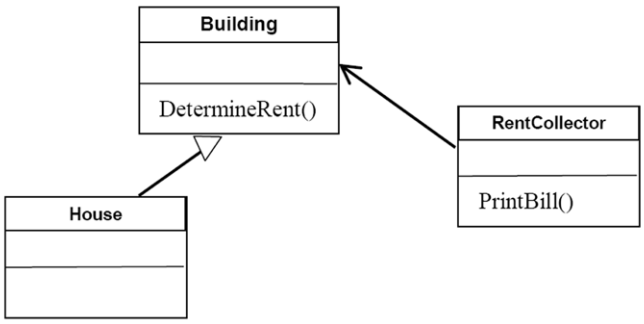
Activity 4

Describe the following using a suitable class diagram showing ANY sensible relationship…

a building for rent
this will have a method to determine the rent
a house for rent
this will inherit the determine rent method
a rent collector (person)
this person will use the determine rent method to print out a bill

HINT: You may wish to use the following arrow 

Feedback 4



Note: RentCollector does not inherit from Building as a RentCollector is a person not a type of Building. However there is a relationship (an association) between RentCollector and Building ie. a RentCollector needs to determine the rent for a Building in order to print out the bill.

Activity 5

Looking at the feedback from Activity 4 and determine if a RentCollector can print out a bill for the rent due on a house (or can they just print a bill for buildings?).

Feedback 5

Firstly to print out a bill a RentCollector would need to know the rent due. There is no method DetermineRent() defined for a house – but this does not mean it does not exist.

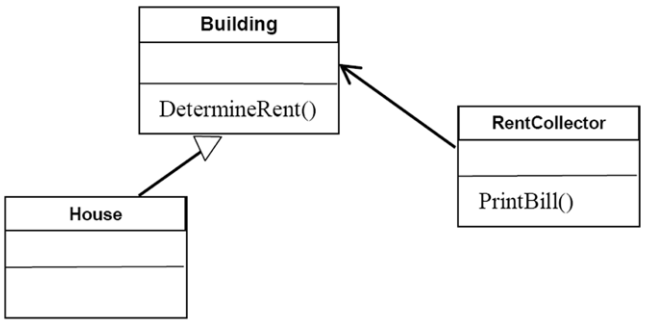
It must exist as House inherits the properties and methods of Building!

We only show methods in subclasses if they are either additional methods or methods that have been overridden.

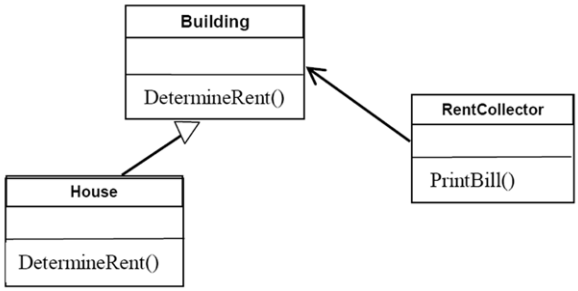
A rent collector requires a building but a House **is a** type of a Building. So, while no association is shown between the RentCollector and House, a Rentcollector could print a bill for a house. Wherever a Building object is required we could substitute a House object as this is a type of Building. This is an example of polymorphism and we will see other examples of this in Chapter 4.

Activity 6

Modify this UML diagram to show that DetermineRent() is overridden, i.e. replaced, in House.



Feedback 6



By showing DetermineRent() in House we are showing that this method is overriding the one defined in the base class (Building).

Interestingly the .NET CLR engine will use the most correct DetermineRent() method depending upon which type of object the method is invoked on. Thus RentCollector will invoke the method defined in House if printing a bill for a house but will use the method defined in Building for any other type of building. This is automatic – the code in the RentCollector class does not distinguish between different types of Building.

Overriding will be discussed in more detail later in this chapter.

3.4 Implementing Inheritance in C#

No special features are required to create a superclass. Thus any class can be a superclass unless specifically prevented.

A subclass specifies it is inheriting features from a superclass using the : **symbol**. For example….

```
 class MySubclass : MySuperclass
{
     // additional instance variables and
     // additional methods
}
```

3.5 Constructors

Constructors are methods that create objects from a class. Each class (whether sub or super) should encapsulate its own initialization in a constructor, usually relating to setting the initial state of its instance variables. Constructors are methods given the same name as the class.

A constructor for a superclass, or base class, should deal with general initialization.

Each subclass can have its own constructor for specialised initialization but it must often invoke the behaviour of the base constructor. It does this using the keyword **base.**

```
class MySubClass : MySuperClass
{
     public MySubClass (sub-parameters) : base(super-parameters)
     {
           // other initialization
     }
```

Usually some of the parameters passed to MySubClass will be initializer values for superclass instance variables, and these will simply be passed on to the superclass constructor as parameters. In other words *super-parameters* will be some (or all) of *sub-parameters*.

Shown below are two constructors, one for the Publication class and one for Book. The book constructor requires four parameters three of which are immediately passed on to the base constructor to initialize its instance variables.

```
// a constructor for the Publication class
public Publication(String title, double price, int copies)
{
     this.title = title;
     // etc
}
```

```
// a constructor for the Book class
public Book(String title, double price, int copies, String author)
          : base(title, price, copies)
{
     this.author = author;
}
```

Thus in creating a book object we first create a publication object. The constructor for Book does this by calling the constructor for Publication.

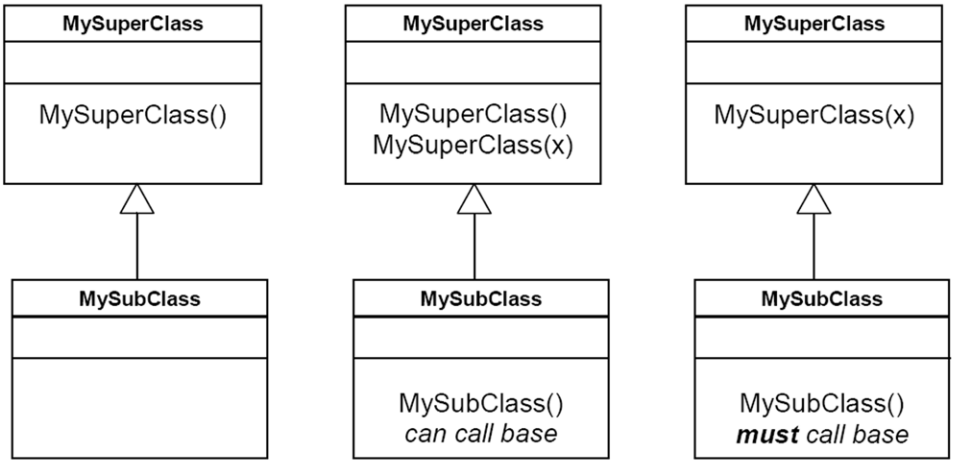
3.6 Constructor Rules

Rules exist that govern the invocation of a superconstructor.

If the superclass has a parameterless (or default) constructor this will be called automatically if no explicit call to base is made in the subclass constructor though an explicit call is still better style for reasons of clarity.

However if the superclass has no parameterless constructor but does have a parameterized one, this **must** be called explicitly using : base.

To illustrate this….



On the left above:- it is legal, though bad practice, to have a subclass with no constructor because superclass has a parameterless constructor.

In the centre:- if subclass constructor doesn't call the base constructor then the parameterless superclass constructor will be called.

On the right:- because superclass has no paramterless constructor, subclass **must** have a constructor, it **must** call the super constructor using the keyword base and it must pass on the required paramter. This is simply because a (super) class with only a parameterized constructor can only be initialized by providing the required parameter(s).

3.7 Access Control

To enforce encapsulation we normally make instance variables **private** and provide accessor/mutator methods as necessary (or in C# we use properties).

The SellCopy() method of Publication needs to alter the value of the variable 'copies' it can do this even if 'copies' is a private variable. However Book and Magazine both need to alter 'copies'.

There are three ways we can do this in C# …

1) make 'copies' 'protected' rather than 'private' – this makes it visible to subclasses, **or**
2) create accessor and mutator methods.
3) create 'properties' in C#. These are effectively accessor methods but make the coding simpler.

Generally we should keep variables private and create accessors/mutators methods rather than compromise encapsulation, though **protected** may be useful to allow subclasses to use methods (e.g. accessors and mutators) which we would not want generally available to other classes. In C# it is simpler and hence normal practise to create properties which effectively do the same job as accessor methods.

We will show demonstrate the use of accessor methods and properties here.

Firstly using accessor methods: In the superclass Publication we define 'copies' as a variable private but create two methods that can set and access the value 'copies'. As these accessor methods are public or protected they can be used within a subclass when access to 'copies' is required.

In the superclass Publication we would therefore have….

```
private int copies;
public int GetCopies ()
{
     return copies;
}
public void SetCopies(int newValue)
{
     copies = newValue;
}
```

These methods allow the superclass to control access to private instance variables.

As currently written they don't actually impose any restrictions, but suppose for example we wanted to make sure 'copies' is not set to a negative value.

(a) If 'copies' is **private**, we can put the validation (i.e. an if statement) within the SetCopies() method here and know for sure that the rule can never be compromised.

(b) If 'copies' is partially exposed as **protected**, we would have to look at every occasion where a subclass method changed the instance variable and do the validation at each separate place.

We might even consider making these *methods* **protected** rather than **public** themselves so their use is restricted to subclasses only and other classes cannot interfere with the value of 'copies' at all.

Making use of these methods in the sublasses Book and Magazine we have ..

```
// in Book
public void OrderCopies(int orderQty)
{
     SetCopies(GetCopies() + orderQty);
}
```

```
// and in Magazine
public void RecNewIssue(String newIssue)
{
     SetCopies(orderQty);
     currIssue = newIssue;
}
```

These statements are equivalent to

copies = copies + orderQty;

and in 'Magazine'

copies = orderQty;

In C# 'properties' can be defined that are really hidden accessor methods and make this code simpler. Here the word 'property' is used with a meaning particular to C# and is not the same as a 'property' of a class.

In the code below two variables are defined, 'price' and 'copies' and a property is defined for each … the properties have the same name as the variable but start with an uppercase letter.

```
private double price;
public double Price
{
     get { return price; }
     set { price = value; }
}
private int copies;
public int Copies
{
     get { return copies; }
     set { copies = value; }
}
```

Thus when we refer to 'copies' we are referring to a private variable that cannot be accessed outside of the class. When we refer to 'Copies' with a capital C we are referring to the public property .. as this is public we can use this to obtain or change the value of 'copies' from any class.

In the code above the properties have been defined such that they will both get and set the value of their respective variables… with no restrictions. We could change this code to impose restrictions or to remove either the 'get' or 'set' method.

By using 'Copies = orderQty' we are effectively invoking a setter method but we are doing this by using the property. This is effectively the same as using the setter method shown earlier to set a new value… 'SetCopies(orderQty);

Thus using the properties we could replace the methods shown above with those shown below.

```
// in Book
public void OrderCopies(int orderQty)
{
     Copies = Copies + orderQty;
}
```

```
// and in Magazine
public void RecNewIssue(String newIssue)
{
     Copies = orderQty;
     currIssue = newIssue;
}
```

Using 'properties' is normal in C#. In other object oriented languages the use of accessor methods does exactly the same job.

3.8 Abstract Classes

The idea of a Publication which is not a Book or a Magazine is meaningless, just like the idea of a Person who is neither a MalePerson nor a FemalePerson. Thus while we are happy to create Book or Magazine objects we may want to prevent the creation of objects of type Publication.

If we want to deal with a new type of Publication which is genuinely neither Book nor Magazine – e.g. a Newspaper – it would naturally become another new subclass of Publication.

As Publication will never be instantiated, ie. we will never create objects of this type, the only purpose of the class exists is to gather together the generalized features of its subclasses in one place for them to inherit.

We can enforce the fact that Publication is non-instantiable by declaring it 'abstract':-

```
public abstract class Publication
{
// etc.
```

3.9 Overriding Methods

A subclass inherits the methods of its superclass and must therefore always provide at least that set of methods, and often more. However, the implementation of a method can be changed in a subclass.

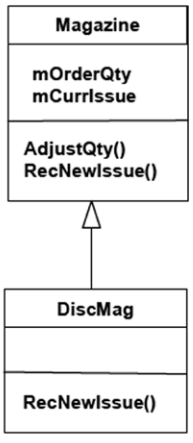
This is overriding the method.

To do this we write a new version in the subclass which replaces the inherited one.

The new method should essentially perform the same functionality as the method that it is replacing however by changing the functionality we can improve the method and make its function more appropriate to a specific subclass.

For example, imagine a special category of magazine which has a disc attached to each copy – we can call this a DiscMag and we would create a subclass of Magazine to deal with DiscMags. When a new issue of a DiscMag arrives not only do we want to update the current stock but we want to check that the discs are correctly attached. Therefore we want some additional functionality in the RecNewIssue() method to remind us to do this. We achieve this by redefining RecNewIssue() in the DiscMag subclass.

Note: when a new issue of Magazine arrives, as these don't have a disc we want to invoke the original RecNewIssue() method defined in the Magazine class.



- The definition of **RecNewIssue()** in DiscMag overrides the inherited one.
- Magazine is not affected – it retains its original definition of **RecNewIssue()**
- By showing **RecNewIssue()** in DiscMag we are stating that the inherited method is being overridden (ie. replaced) as we do not show in inherited methods in subclasses.

When we call the **RecNewIssue**() method on a DiscMag object the CLR engine automatically selects the new overriding version – the caller doesn't need to specify this, or even know that it is an overriden method at all. When we call the **RecNewIssue**() method on a Magazine it is the method in the superclass that is invoked.

Implementing DiscMag

To implement DiscMag we must create a subclass of Magazine. No additional instance variables or methods are required though it is possible to create some if there was a need. The constructor for DiscMag simply passes ALL its parameters directly on to the superclass and a version of RecNewIssue() is defined in DiscMag to override the one inherited from Magazine (see code below).

```
public class DiscMag : Magazine
{
     public DiscMag(String title, double price, int copies, int orderQty, String currIssue)
          : base(title, price, copies, orderQty, currIssue)
     {
     }

     public override void RecNewIssue(String newIssue)
     {
          base.RecNewIssue (newIssue);
          Console.WriteLine("Check discs are attached");
     }
}
```

Note the use of base.RecNewIssue() to call a method of the superclass, thus re-using the existing functionality as part of the replacement, just as we do with constructors. It then additionally displays the required message for the user.

One final change is required. Before a method can be overridden permission for this must be granted by the author of the superclass. Using the keyword virtual when defining methods basically grants permission for them to be overridden.

Thus for the code above to work the RecNewIssue() method in the magazine must be made virtual. See below…

```
// in Magazine
public virtual void RecNewIssue(String newIssue)
{
     Copies = orderQty;
     currIssue = newIssue;
}
```

This mechanism gives us the ability to allow, or prevent, the methods we create from being overridden in subclasses.

Operations

Formally, 'RecNewIssue()' is an operation. This one operation is implemented by two different methods, one in Magazine and the overriding one in DiscMag. This distinction is an important part of 'polymorphism' which we will meet in Chapter 4.

3.10 The 'Object' Class

In C# all classes are (direct or indirect) subclasses of a class called 'Object'. Object is the 'root' of the inheritance hierarchy in C#. Thus this class exists in every C# program ever created.

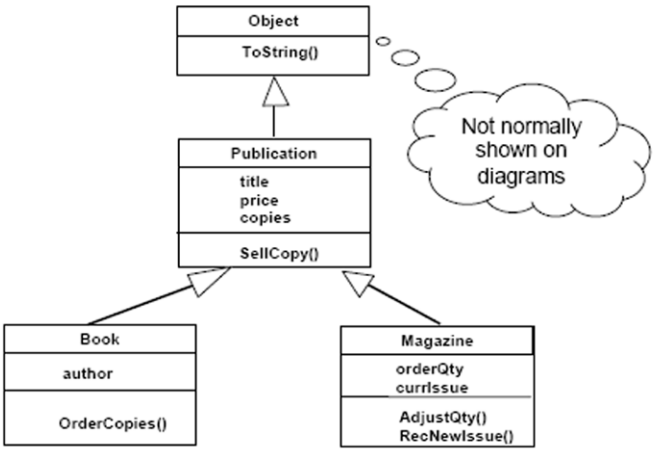
If a class is not declared to inherit from another then it implicitly inherits from Object.

'Object' defines no instance variables but several methods. Generally these methods will be overridden by new classes to make them useful. An example is the ToString() method.

Thus when we define our own classes, by default they are direct subclasses of Object.

If our classes are organised into a hierarchy then the topmost superclass in the hierarchy is a direct subclass of object, and all others are indirect subclasses.

Thus directly, or indirectly, all classes created in C# inherit the ToString() method defined in the Object class.



3.11 Overriding ToString() defined in 'Object'

The Object class defines a ToString() method, one of several useful methods.

ToString() has the signature

public String ToString()

The purpose of the ToString() method is to return a string value that represents the current object. The version of ToString() defined by Object produces output like: "Book". This is the class name from which an object is instantiated. However to be generally useful we need to override this to give a more meaningful string.

In Publication

```
public override String ToString()
{
     return title;
}
```

In Book

```
public override String ToString()
{
     return base.ToString() + " by " + author;
}
```

In Magazine

```
public override String ToString()
{
     return base.ToString() + " (" + currIssue + ")";
}
```

In the code above ToString() originally defined in Object has been completely replaced, ie. overridden, so that Publication. ToString() returns the title of the publication.

The ToString() method has been overridden again in Book such that Book.ToString() returns title (via the base classes' ToString() method) and author i.e. this overridden version uses the version defined in Publication. Thus if Publication. ToString() was rewritten to return the title and ISBN number then Book.ToString() would automatically return the title, ISBN number and author.

Magazine.ToString() returns title (via the base class ToString() method) and issue.

We will do not need to further override the method in DiscMag because the version it inherits from Magazine is OK.

We could choose to provide more data (i.e. more, or even all, of the instance variable values) in these strings. The design judgement here is that these will be the most generally useful printable representation of objects of these classes. In this case title and author for a book, or title and current issue for a magazine, serve well to uniquely identify a particular publication.

Perhaps for a Newspaper we would override ToString() to return the title of the newspaper and the date it was printed.

3.12 Summary

Inheritance allows us to factor out common attributes and behaviour. We model the commonalities in a superclass (sometimes called a base class in C#).

Subclasses are used to model specialized attributes and behaviour.

Code in a superclass is inherited by all subclasses. If we amend or improve code for a superclass it impacts on all subclasses. This reduces the code we need to write in our programs.

Special rules apply to constructors for subclasses.

A superclass can be declared abstract to prevent it being instantiated (i.e. objects created).

We can 'override' inherited methods so a subclass implements an operation differently from its superclass.

In C# all classes descend from the base class 'Object'

'Object' defines some universal operations which can usefully be overriden in our own classes.

4 Object Roles and the Importance of Polymorphism

Introduction

Through the use of worked examples this chapter will explain the concept of polymorphism and the impact this has on OO software design.

Objectives

By the end of this chapter you will be able to….

- Understand how polymorphism allows us to handle related classes in a generalized way
- Employ polymorphism in C# programs
- Understand the implications of polymorphism with overridden methods
- Define interfaces to extend polymorphism beyond inheritance hierarchies
- Appreciate the scope for extensibility which polymorphism provides

This chapter consists of eight sections :-

1) Class Types
2) Substitutability
3) Polymorphism
4) Extensibility
5) Interfaces
6) Extensibility Again
7) Distinguishing Subclasses
8) Summary

4.1 Class Types

Within hierarchical classification of animals

Pinky is a pig (species sus scrofa)
Pinky is (also, more generally) a mammal
Pinky is (also, even more generally) an animal

We can specify the type of thing an organism is at different levels of detail:

higher level = less specific
lower level = more specific

If you were asked to give someone a pig you could give them Pinky or any other pig.

If you were asked to give someone a mammal you could give them Pinky, any other pig or any other mammal (e.g. any lion, or any mouse, or any cat).

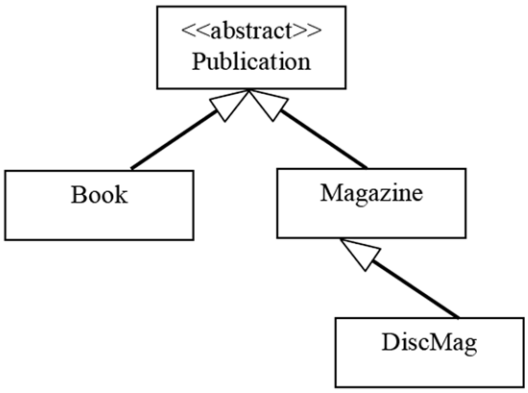
If you were asked to give someone an animal you could give them Pinky, any other pig, any other mammal, or any other animal (bird, fish, insect etc).

The idea here is that an object in a classification hierarchy has an 'is a' relationship with every class from which it is descended and each classification represents a type of animal.

This is true in object oriented programs as well. Every time we define a class we create a new 'type'. Types determine compatibility between variables, parameters etc.

A subclass type is a subtype of the superclass type and we can substitute a subtype wherever a 'supertype' is expected. Following this we can substitute objects of a subtype whenever objects of a supertype are required (as in the example above).

The class diagram below shows a hierarchical relationship of types of object – or classes.



- If we want 'a DiscMag', it must be an object of class DiscMag.
- If we want 'a Magazine', it could be an object of class Magazine or an object of class DiscMag (as this is a type of magazine).
- If we want 'a Publication' it could be a Book, Magazine or DiscMag.

In other words we can 'substitute' an object of any subclass where an object of a superclass is required. This is NOT true in reverse!

Activity 1

In C# the keyword new invokes the constructor of a class i.e. creates an object of that class. Look at the class diagram above and decide which of the following lines of code would be legal in a C# program where these classes had been implemented: -

Publication p = new Book(…);

Publication p = new DiscMag(…);

Magazine m = new DiscMag(…);

DiscMag dm = new Magazine(…);

Publication p = new Publication(…);

Feedback 1

Publication p = new Book(…);

Here we are defining a variable p of the general type of 'Publication' we are then invoking the constructor for the Book class and assigning the result to 'p' this is OK because Book is a subclass of Publication i.e. a Book is a Publication.

Publication p = new DiscMag(…);

This is OK because DiscMag is a subclass of Magazine which is a subclass of Publication ie. DiscMag is an indirect subclass of Publication.

Magazine m = new DiscMag(…);

This is OK because DiscMag is a subclass of Magazine

DiscMag dm = new Magazine(…);

This is illegal because Magazine is a SUPERclass of DiscMag. Some Magazines are DiscMags but some are not so if a DiscMag is required we cannot hand over any Magazine.

Publication p = new Publication(…);

This is illegal for a different reason.. Publication is an abstract class and therefore cannot be instantiated.

4.2 Substitutability

When designing class/type hierarchies, the type mechanism allows us to place a subclass object where a superclass is specified. However this has implications for the design of subclasses – we need to make sure they are genuinely substitutable for the superclass. If a subclass object is substitutable then clearly it must implement all of the methods of the superclass – this is easy to guarantee as all of the methods defined in the superclass are inherited by the subclass. Thus while a subclass may have additional methods it must at least have all of the methods defined in the superclass and should therefore be substitutable. However what happens if a method is overridden in the subclass?

When overriding methods we must ensure that they are still substitutable for the method being replaced. Therefore when overriding methods, while it is perfectly acceptable to tailor the method to the needs of the subclass a method should not be overridden with functionality which performs an inherently different operation.

For example, RecNewIssue() in DiscMag overrides RecNewIssue() from Magazine but does the same basic job ("fulfils the contract") as the inherited version with respect to updating the number of copies and the current issue. While it extends that functionality in a way specifically relevant to DiscMags by displaying a reminder to check the cover discs, essentially these two methods perform the same operation.

What do we know about a 'Publication'?

Answer: It's an object which supports (at least) the operations:

void SellCopy()
String ToString()
and it has properties that allow us to
set the price,
get the number of copies
set the number of copies.

Inheritance guarantees that objects of any subclass of Publications provides at least these.

Note that a subclass can never remove an operation inherited from its superclass(es) – this would break the guarantee. Because subclasses extend the capabilities of their superclasses, the superclass functionality can be assumed.

It is quite likely that we would choose to override the ToString() method (initially defined within 'Object') within Publication and override it again within Magazine so that the String returned provides a better description of Publications and Magazines. However we should not override the ToString() method in order to return the price – this would be changing the functionality of the method so that the method performs an inherently different function. Doing this would break the substitutability principle.

4.3 Polymorphism

Because an instance of a subclass is an instance of its superclass we can handle subclass objects as if they were superclass objects. Furthermore because a superclass guarantees certain operations in its subclasses we can invoke those operations without caring which subclass the actual object is an instance of.

This characteristic is termed 'polymorphism', originally meaning 'having multiple shapes'.

Thus a Publication comes in various shapes … it could be a Book, Magazine or DiscMag. We can invoke the SellCopy() method on any of these Publications irrespective of their specific details.

Polymorphism is a fancy name for a common idea. Someone who knows how to drive can get into and drive most cars because they have a set of shared key characteristics – steering wheel, gear stick, pedals for clutch, brake and accelerator etc – which the driver knows how to use. There will be lots of differences between any two cars, but you can think of them as subclasses of a superclass which defines these crucial shared 'operations'.

If 'p' 'is a' Publication, it might be a Book or a Magazine or a DiscMag.

Whichever it is we know that it has a SellCopy() method.

So we can invoke p.SellCopy() without worrying about what exactly 'p' is.

This can make life a lot simpler when we are manipulating objects within an inheritance hierarchy. We can create new types of Publication e.g. a Newspaper and invoke p,SellCopy() on a Newspaper without have to create any functionality within the new class – all the functionality required is already defined in Publication.

Polymorphism makes it very easy to extend the functionality of our programs as we will see now and we will see this again in the case study (in Chapter 11).

4.4 Extensibility

Huge sums of money are spent annually creating new computer programs but over the years even more is spent changing and adapting those programs to meet the changing needs of an organisation. Thus as professional software engineers we have a duty to facilitate this and help to make those programs easier to maintain and adapt. Of course the application of good programming standards, commenting and layout etc, have a part to play here but also polymorphism can help as it allows programs to be made that are easily extended.

CashTill class

Imagine we want to develop a class CashTill which processes a sequence of items being sold. Without polymorphism we would need separate methods for each type of item:

SellBook (Book pBook)
SellMagazine (Magazine pMagazine)
SellDiscMag (DiscMag pDiscMag)

With polymorphism we need only
SellItem (Publication pPub)

Every subclass is 'type-compatible' with its superclass. Therefore any subclass object can be passed as a Publication parameter.

This also has important implications for extensibility of systems. We can later introduce further subclasses of Publication and these will also be acceptable by the SellItem() method of a CashTill object, even through these subtypes were unknown when the CashTill was implemented.

Publications sell themselves!

Without polymorphism we would need to check for each item 'p' so we were calling the right method to sell a copy of that subtype

if 'p' is a Book call SellCopy() method for Book
else if 'p' is a Magazine call SellCopy() method for Magazine
else if 'p' is a DiscMag call SellCopy() method for DiscMag

Instead we trust C# to look at the object 'p' at run time, to determine its 'type' and its own method for selling itself. Thus we can call :-
p.SellCopy()

and if the object is a Book it will invoke the SellCopy() method for a Book. If 'p' is a Magazine, again at runtime C# will determine this and invoke the SellCopy() method for a Magazine.

Polymorphism often allows us to avoid conditional 'if' statements – instead the 'decision' is made implicitly according to which type of subclass object is actually present.

Implementing CashTill

The code below shows how CashTill can be implemented to make use of Polymorphism.

```
public class CashTill
{
     private double runningTotal;
     public CashTill()
     {
          runningTotal = 0;
     }
     public void SellItem(Publication pPub)
     {
          runningTotal = runningTotal + pPub.Price;
          pPub.SellCopy();
          Console.WriteLine("Sold " + pPub + " @ " +
                  pPub.Price + "\nSubtotal = " +
                  runningTotal);
     }
     public void ShowTotal()
     {
          Console.WriteLine("GRAND TOTAL: " + runningTotal);
     }
}
```

The CashTill has one instance variable – a double to hold the running total of the transaction. The constructor simply initializes this to zero.

The SellItem() method is the key feature of CashTill. It takes a Publication parameter, which may be a Book, Magazine or DiscMag. First the price of the publication is added to the running total using the Price property defined in the class Publication. Then the SellCopy() operation is invoked on the publication.

Finally a message is constructed and displayed to the user, e.g.

Sold Windowcleaning Weekly (Sept 2005) @ 2.75
Subtotal = 2.75

Note that when pPub appears in conjunction with the string concatentation operator '+'. This implicitly invokes the ToString() method for the subclass of this object, and remember that ToString() is different for books and magazines.

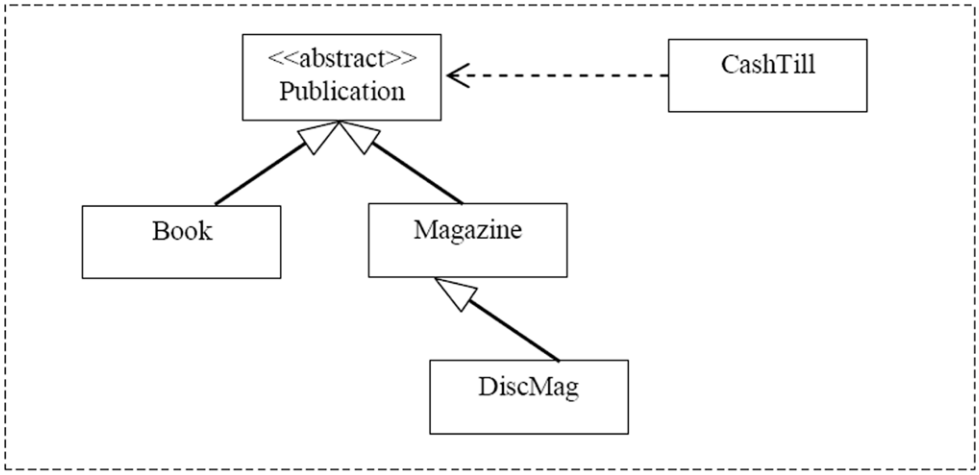
The correct ToString() operation is automatically invoked by C# to return the appropriate string description for the specific object sold!

Thus if a book is sold the output would contain the title and author e.g.

Sold Hitch Hikers Guide to the Galaxy by D Adams @ 7.50
Subtotal = 7.50

Thus our cash till can sell any publication of any shape, i.e. any type Book, Magazine or DiscMag, without worrying about any specific features of these classes. This is polymorphism in action!

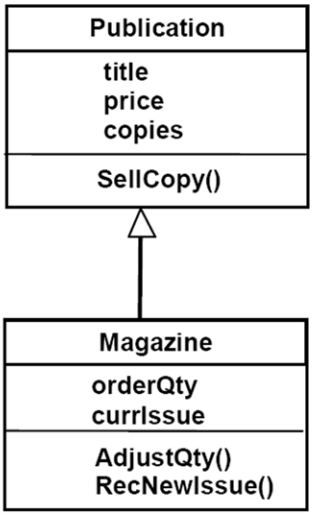
We can show CashTill on a class diagram as below :-



Note that CashTill has a dependency on Publication because the SellItem() method is passed a parameter of type Publication. What is actually passed will of course be an object of one of the concrete types descended from Publication.

Activity 2

Look at the diagram below and, assuming Publication is not an abstract type, decide which of the pairs of operations shown are legal.



a) Publication p = new Publication(…);
p.SellCopy();

b) Publication p = new Publication(…);
p.RecNewIssue();

c) Publication p = new Magazine(…);
p.SellCopy();

d) Publication p = new Magazine(…);
p.RecNewIssue();

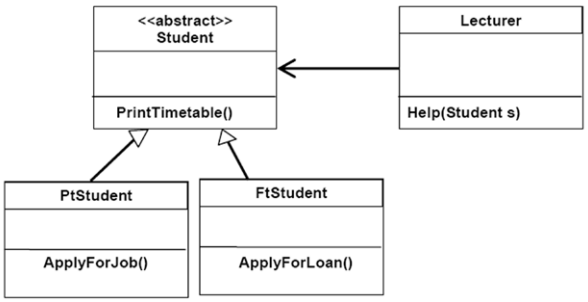
e) Magazine m = new Magazine(…);
m.RecNewIssue();

Feedback 2

a) Legal – you can invoke SellCopy() on a publication
b) Illegal – the RecNewIssue() method does not exist in publications
c) Legal – Magazine is a type of Publication and therefore you can assign an object of type Magazine to a variable of type Publication (you can always substitute subtypes where a supertype is requested). Also you can invoke SellCopy() on a publication. The publication happens to be a magazine but this is irrelevant as far as the compiler knows in this instance 'p' is just a publication.
d) Illegal – while we can invoke RecNewIssue on a magazine the compiler does not know that 'p' is a magazine…only that it is a publication.
e) Legal – m is a magazine and we can invoke this method on magazines.

Activity 3

Look at the diagram below and, noting that Student is an abstract class, decide which of the following code segments are valid….



Note FtStudent is short for Full Time Student and PtStudent is short for Part Time Student.

a) Student s = new Student();
   Lecturer l = new Lecturer();
   l.Help(s);

b) Student s = new FtStudent();
   Lecturer l = new Lecturer();
   l.Help(s);

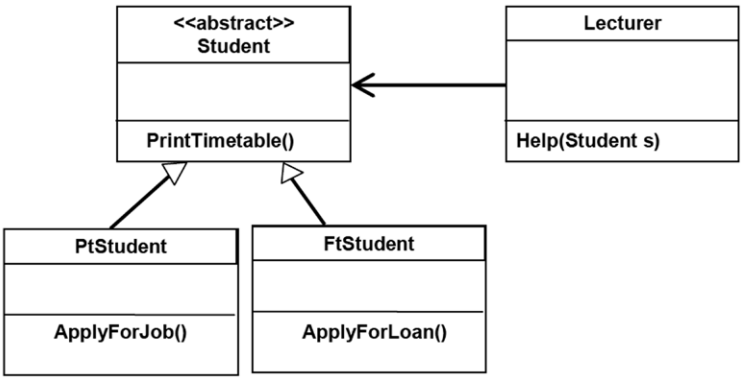
Feedback 3

a) This is not valid as class Student is abstract and cannot be instantiated
b) This is valid. FtStudent is a type of Student and can be assigned to variable of type Student. This can then be passed as a parameter to l.Help()

Activity 4

Taking the same diagram and having invoked the code directly below decide which of the following lines (a) or (b) would be valid inside the method Help(Student s)…

```
Student s = new FtStudent();
Lecturer l = new Lecturer();
l.help(s);
```



a) s.PrintTimetable();
b) s.ApplyForLoan();

Feedback 4

a) This is valid - we can invoke this method on a Student object and also on an FtStudent object (as the method is inherited).
b) Not Valid! While we can invoke this method on a FtStudent object, and we are passing an FtStudent object as a parameter to the Help() method, the Help() method cannot know that the object passed will be a FtStudent (it could be any object of type Student). Therefore there is no guarantee that the object passed will support this method. Hence this line of code would generate a compiler error.

4.5 Interfaces

There are two aspects to inheritance:

- the subclass inherits the interface (i.e. access to public members) of its superclass – this makes polymorphism possible
- the subclass inherits the implementation of its superclass (i.e. instance variables and method implementations) – this saves us copying the superclass details in the subclass definition

In C#, the use of inheritance, via the ':'symbol, automatically applies both these aspects.

A subclass is a subtype. It's interface must include all of the interface of its superclass, though the implementation of this can be different (though overriding) and the interface of the subclass may be more extensive with additional features being added.

However, sometimes we may want two classes to share a common interface without putting them in an inheritance hierarchy. This might be because :-

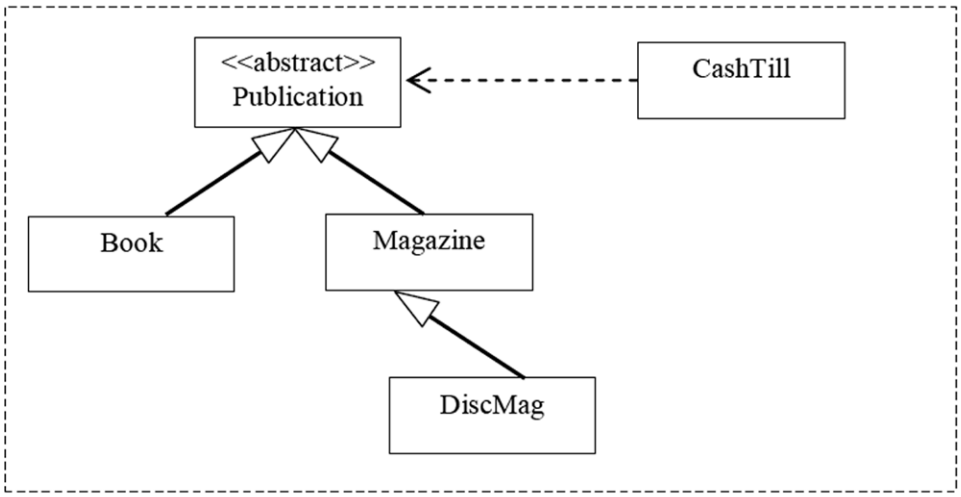
- they aren't really related by a true 'is a' relationship
- we want a class to have interfaces shared with more than one would-be superclass, but C# does not allow such 'multiple inheritance'
- we want to create a 'plug and socket' arrangement between software components, some of which might not even be created at the current time.

This is like making sure that two cars have controls that work in exactly the same way, but leaving it to different engineers to design engines which 'implement' the functionality of the car, possibly in quite different ways.

Be careful of the term 'interface' – in C# programming it has at least three meanings:

1) the public members of a class – the meaning used above
2) the "user interface" of a program, often a "Graphical User Interface" – an essentially unrelated meaning
3) a specific C# construct which we are about to meet

Recall how the subclasses of Publication provide additional and revised behaviour while retaining the set of operations – i.e. the interface – which it defined.



This is why the CashTill class can deal with a 'Publication' without worrying of which specific subclass it is an instance. (Remember that Publication is an abstract class – a 'Publication' is in reality **always** a subclass.)

Tickets

Now consider the possibility that in addition to books and magazines, we now want to sell tickets, e.g. for entertainment events, public transport, etc. These are not like Publications because:-

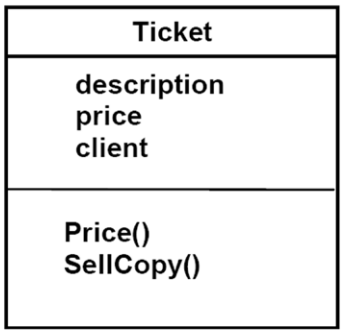
- we don't have a finite 'stock' but print them on demand at the till
- tickets consist simply of a description, price and client (for whom they are being sold)
- these sales are really a service rather than a product

Tickets seem to have little in common with Publications – they share a small **interface** associated with being sold, but even for this the underlying **implementation** will be different because we will not be decrementing them from a current stock.

For these reasons Ticket and Publication do not seem closely related and thus we do not want to put them in an inheritance hierarchy. However we do want to make them both acceptable to CashTill as things to sell and we need a mechanism for doing this.

Without putting them in an inheritance hierarchy what we want is a more general way of saying "things of this class can be sold" which can be applied to whatever (present and future) classes we wish, thus making the system readily extensible to Tickets and anything else.

While the Ticket class is sufficiently different from a Publication that we don't want to put it in an inheritance hierarchy it does have some similarities – namely it has a SellCopy() method and a property to obtain the price – both of these are needed by a CashTill.



However the SellCopy() method is very different form the SellCopy() method defined in Publication. To sell a publication the stock had to be reduced by 1 – with a ticket we just need to print it.

```
public void SellCopy()
{
     Console.WriteLine("*************************************");
     Console.WriteLine("            TICKET VOUCHER            ");
     Console.WriteLine(this.ToString());
     Console.WriteLine("*************************************");
     Console.WriteLine();
}
```

As the SellCopy() method is so different we do not want to inherit its implementation details therefore we don't feel that Ticket belongs in an inheritance hierarchy with Publications. But we do want to be able to check tickets through the till as we can with publications.

Just like publications, tickets provide the operations which CashTill needs:

SellCopy()
Price()

and thus the CashTill can sell a Ticket. In fact CashTill can sell anything that has these methods, not just Publications. To enable this to happen we will define this set of operations as an 'Interface' called ISaleableItem (where 'I' is being used to indicate this refers to an interface not a class).

```
public interface ISaleableItem
{
     double Price
     {
          get;
     }
     void SellCopy();
}
```

Note that the interface defines purely the signatures of operations without their implementations. Note while this interface defines the need for a get method for 'price' a set method is not required and therefore not defined in the interface.

All the methods are implicitly public even if this is not stated, and there can be no instance variables, constructors or code to implement the methods.

In other words, an interface defines the **availability** of specified operations without saying anything about their implementation. That is left to classes which **implement** the interface.

An interface is a sort of contract. The **SaleableItem** interface says "I undertake to provide, at least, methods with these signatures:

```
public void SellCopy ();
public double Price ();
```

though I might include other things as well"

Where more than one class implements an interface it provides a guaranteed area of commonality which polymorphism can exploit.

Think of a car and a driving game in an arcade. They certainly are not related by any "is a" relationship – they are entirely different kinds of things, one a vehicle, the other an arcade game. But they both implement what we could call a "SteeringWheel interface" which we can use in exactly the same way, even though the implementation (mechanical linkage in the car, video electronics in the game) are very different.

We now need to state that both Publication (and all its subclasses) and Ticket both offer the operations defined by this interface:

```
public abstract class Publication : ISaleableItem
{
     [...class details...]
}
```

```
public class Ticket : ISaleableItem
{
     [...class details...]
}
```

In C# the same symbol ':' is used when we define a subclass that extends a super class or when we create a class that implements an interface.

Contras**t implementing an interface** with **extending a superclass.**

- When we extend a superclass the subclass inherits of both interface and implementation from the superclass.
- When we implement an interface we give a guarantee that the operations specified by an interface will be provided – this is enough to allow polymorphic handling of all classes which implement a given interface

The Polymorphic CashTill

The CashTill class already employs polymorphism: the SellItem() method accepts a parameter of type Publication which allows any of its subclasses to be passed:

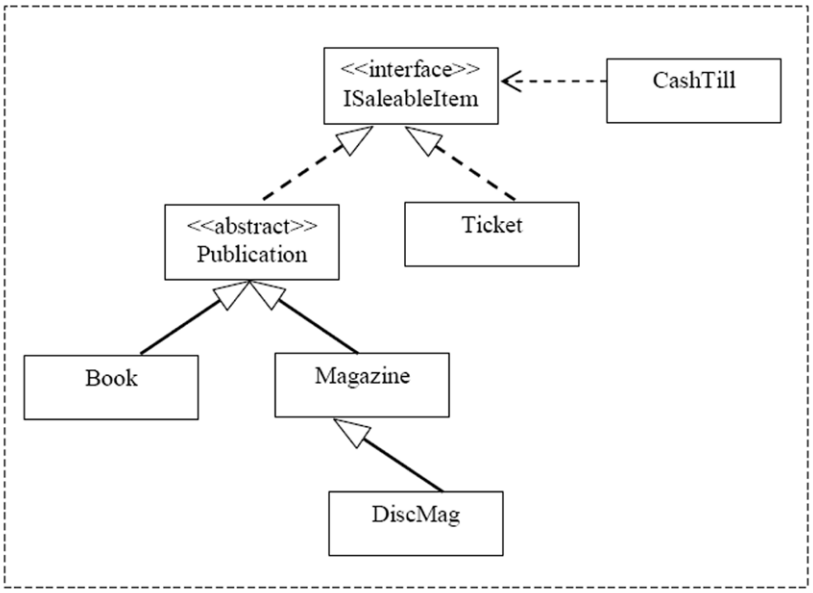
public void SellItem (Publication pPub)

We now want to broaden this further by accepting anything which implements the SaleableItem interface:

public void SellItem(ISaleableItem pSI)

When the type of a variable or parameter is defined as an interface, this works just like a superclass type. Any class which implements the interface is acceptable for assignment to the variable/parameter because the interface is a **type** and all classes implementing it are subtypes of that type.

This is now shown below….



CashTill is no longer directly dependent on class Publication – instead it is dependent on the interface ISaleableItem.

Note we can start the interface name with an 'I'to indicate this is an interface not a class.

The relationships from Publication and Ticket to ISaleableItem are like inheritance arrows except that the lines are **dotted** – this shows that each class **implements** the interface.

A class in C# may only inherit from one superclass but can implement as many interfaces as desirable. The format for this is :-

```
class MyClass : MySuperClass, IMyInterface, IMySecondInterface
```

4.6 Extensibility Again

Polymorphism allows objects to be handled without regard for their precise class. This can assist in making systems extensible without compromising the encapsulation of the existing design.

For example, we could create new classes for more products or services and so long as they implement the SaleableItem interface the CashTill will be able to process them w**ithout a single change to its code**!

An example could be 'Sweets'. We could define a class Sweets to represent sweets in a jar. We can define the price of the sweets depending upon the weight and then sell the sweets by subtracting this weight from our total stock. This is not like selling a Publication, where we always subtract 1 from the stock. Nor is this like selling tickets, where we just print them.

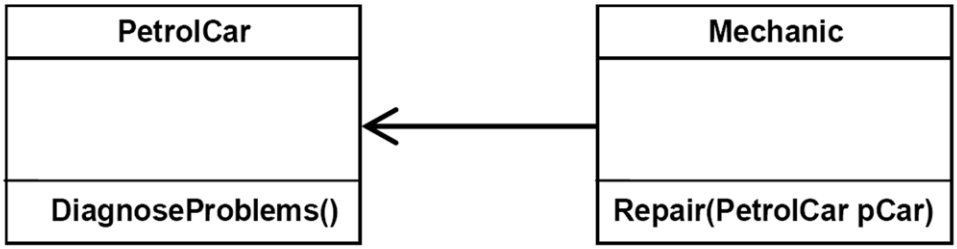
However if we create a class 'Sweets' that implements the ISaleableItem interface our enhanced polymorphic cash till can sell them because it a sell any saleable item.

In this case, without polymorphism we would need to add an additional 'sale' method to CashTill to handle Tickets, Sweets and further new methods for every new type of product to be sold. By defining the ISaleableItem interface can introduce additional products without affecting CashTill at all. Poymorphism makes it easy to extend our programs and this is very important as it saves effort, time and money.

Interfaces allow software components to plug together more flexibly and extensibly, just as many other kinds of plugs and sockets enable audio, video, power and data connections in the everyday world. Think of the number of different electrical appliances which can be plugged into a standard power socket – and imagine how inconvenient it would be if instead you had to call out an electrician to wire up each new one you bought!

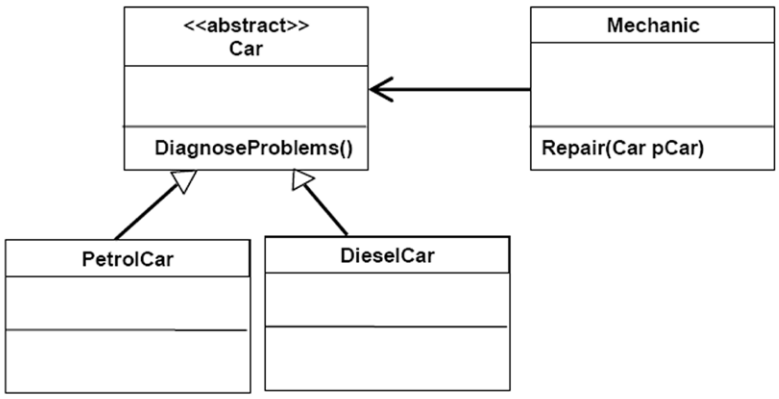
Activity 5

Adapt the following diagram by adding a class for Diesel cars in such a way that it can be used to illustrate polymorphism.



Feedback 5

This is one solution to this exercise… there are of course others.



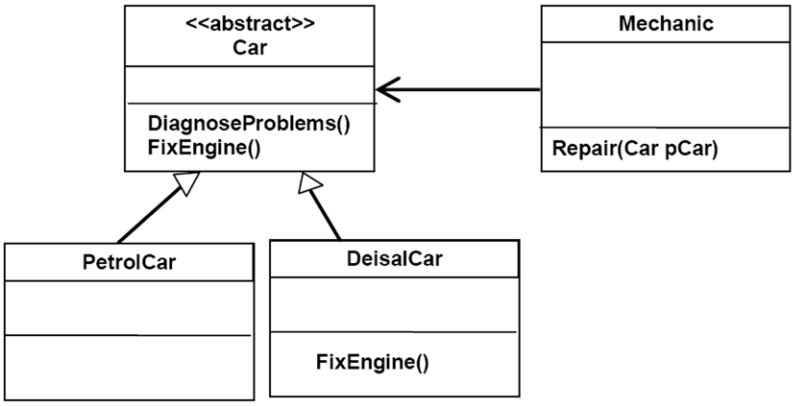
Here Mechanic is directly interacting with Car. In doing so it can interact with any subtype of Car e.g. Petrol, Diesel or any other type of Car developed in the future e.g. (Electric). These are all different (different shapes – at least different internally) and yet Mechanic can still interact with them as they are all Cars. This is polymorphic.

If an ElectricCar class was added Mechanic would still be able to work with them without making any changes to the Mechanic class.

Activity 6

Assume Car has a FixEngine() method that is overridden in DieselCar but not overridden in PetrolCar (as shown on the diagram below).

Look at this diagram and answer the following questions…



a) Would the following line of code be valid inside the Repair() method ?
pCar.FixEngine();

b) If a DiesalCar object was passed to the repair() method which actual method would be invoked by pCar. FixEngine(); ?

Feedback 6

Yes! We can apply the method FixEngine() to any Car object as it is defined in the class Car.

This would invoke the overridden method. The method must be defined in the class Car else the compiler will complain at compile time. However at run time the identity the actual object passed will be checked. As the actual object is a subtype DiesalCar the actual method invoked will be the overridden method. Clever stuff given that the Repair() method is unaware of which type of car is actually passed!

4.7 Distinguishing Subclasses

What if we have an object handled polymorphically but need to check which subtype it **actually** is? The is operator can do this:

object is *class*

This test is **true** if the object is of the specified class (or a subclass), **false** otherwise.

Note that (**myDiscMag is Magazine)** would be TRUE because a DiscMag is a Magazine

is can also be used with an interface name on the right, in which case it tests whether the class implements the interface.

Strictly is is testing whether the item on the left is of the type, or a subtype of, the **type** specified on the right. Doing this we could extend the CashTill class such that it displays a specific message depending upon the object sold.

```
public void SaleType (ISaleableItem pSI)
{
     if (pSI is Publication)
     {
          Console.WriteLine("This is a Publication");
     }
      else if (pSI is Ticket)
     {
          Console.WriteLine ("This is a Ticket");
     }
     else
     {
          Console.WriteLine ("This is a an unknown sale type");
     }
}
```

pIS is Publication will be true if pIS is any subclass of Publication (i.e. a Book, Magazine or DiscMag). If we wished to we could equally test for a more specific subtype, e.g. **pIS is Book**

Notice that once we compromise the polymorphism by checking for subtypes we also compromise the extensibility of the system – new classes (e.g. Sweets) implementing the SaleableItem interface may also require new clauses adding to this if statement, so the change ripples through the system with the consequence that it becomes more costly and error-prone to maintain.

Instead of doing this we should try to package different behaviours into the subclasses themselves, e.g. we could define a **DescribeSelf()** method in the interface SaleableItem this would then need to be implemented in each class that implements the SaleableItem interface. Thus each subtype would display a message giving the type of item being sold. The if statement above, in CashTill, can then be replaced with **pIS.DescribeSelf()**. Thus when we add new classes to the system we would not need to change the CashTill class.

4.8 Summary

Polymorphism allows us to refer to objects according to a superclass rather than their actual class.

Polymorphism makes it easy to extend our programs by adding additional classes without needing to change other classes.

We can manipulate objects by invoking operations defined for the superclass without worrying about which subclass is involved in any specific case.

C# ensures that the appropriate method for the actual class of the object is invoked at run-time.

Sometimes we want to employ polymorphism without all the classes concerned having to be in an inheritance hierarchy. An interface allows us to provide shared collections of operations in this situation. When doing this there is no inherited implementation – each class must implement ALL the operations defined by the Interface.

Any number of classes can implement a particular interface.

A class in C# may only inherit from one superclass but can implement multiple interfaces.

5 Overloading

Introduction

This chapter will introduce the reader to the concept of method overloading

Objectives

By the end of this chapter you will be able to….

- Understand the concept of 'overloading'
- Appreciate the flexibility offered by overloading methods
- Identify overloaded methods in the online API documentation

This chapter consists of the following three sections :-

1) Overloading
2) Overloading To Aid Flexibility
3) Summary

5.1 Overloading

Historically in computer programs method names were required to be unique. Thus the compiler could identify which method was being invoked just by looking at its name.

However several methods were often required to perform very similar functionality for example a method could add two integer numbers together and another method may be required to add two floating point numbers. If you have to give these two methods unique names which one would you call 'Add()'?

In order to give each method a unique name the names would need to be longer and more specific. We could therefore call one method AddInt() and the other AddFloat() but this could lead to a proliferation of names each one describing different methods that are essentially performing the same operation i.e. adding two numbers.

To overcome this problem in C# you are not required to give each method a unique name – thus both of the methods above could be called Add(). However if method names are not unique the C# must have some other way of deciding which method to invoke at run time. i.e. when a call is made to Add(number1, number2) the machine must decide which of the two methods to use. It does this by looking at the parameter list.

While the two methods may have the same name they can still be distinguished by looking at the parameter list. :-

```
Add(int number1, int number2)
Add(float number1, float number2)
```

This is resolved at run time by looking at the method call and the actual parameters being passed. If two integers are being passed then the first method is invoked. However if two floating point numbers are passed then the second method is used.

Overloading refers to the fact that several methods may share the same name. As method names are no longer uniquely identify the method then the name is 'overloaded'.

5.2 Overloading To Aid Flexibility

Having several methods that essentially perform the same operation, but which take different parameter lists, can lead to enhanced flexibility and robustness in a system.

Imagine a University student management system. A method would probably be required to enrol, or register, a new student. Such a method could have the following signature …

```
EnrollStudent(String name, String address, String coursecode)
```

However if a student had just arrived in the city and had not yet sorted out where they were living would the University want to refuse to enrol the student? They could do so but would it not be better to allow such a student to enrol (and set the address to 'unkown')?

To allow this the method EnrollStudent() could be overloaded and an alternative method provided as…

EnrollStudent(String name, String coursecode)

At run time the method invoked will depend upon the parameter list provided. Thus given a call to

EnrollStudent("Fred", "123 Abbey Gardens", "G700")

the first method would be used.

Activity 1

Imagine a method WithdrawCash() that could be used as part of a banking system. This method could take two parameters :- the account identity (a String) and the amount of cash required by the user (int). Thus the full method signature would be :-

WithdrawCash(String accountID, int amount).

Identify another variation of the WithdrawCash() method that takes a different parameter list that may be a useful variation of the method above.

Feedback 1

An alternative method also used to withdraw cash could be WithdrawCash(String accountID) where no specified amount is provided but by default £100 is withdrawn.

These methods essentially perform the same operation but by overloading this method we have made the system more flexible – users now have a choice they can specify the amount of cash to be withdrawn or they can accept the default sum specified.

Overloading methods don't just provide more flexibility for the user they also provide more flexibility for programmers who may have the job of extending the system in the future and thus overloading methods can make the system more future proof and robust to changing requirements.

Constructors can be overloaded as well as ordinary methods.

Activity 2

Go online to msdn.microsoft.com. This is the website for the Microsoft Developer Network and the contains details of the Application Programmer Interface (API) for the .NET framework (it also contains much more).

Follow the links for
Library (tab near top of page)
.NET Development (on left pane)
.NET Framework 4
.NET Framework Class Library

At this point you will see all of the namespaces (packages) in the .NET framework on the left with a description of each on the right)

Follow the link for
System (one of the core namespaces) and then
String (one of the core classes)

The directly link for this is
http://msdn.microsoft.com/en-us/library/system.string.aspx

Select the C# syntax version… remembering that the .NET framework API is available to programmers in several languages.

Look at the String class documentation in the main pane and find out how many constructors exist for this class.

Feedback 2

In .NET version 4 the String class specifies 8 different constructors. They all have the same method name 'String' of course but they can all be differentiated by the different parameters these methods require.

One of these constructors takes and array of characters, a starting position and a length and creates a String object using this subset of characters taken from this array. Another requires a as a parameter a pointer to an array of character and creates a new String object that is a copy of the original.

By massively overloading the Sting constructor the creators of this class have provided flexibility for other programmers who may wish to use these different options in the future.

We can make our programs more adaptable by overloading constructors and other methods. Even if we don't initially use all of the different constructors, or methods, by providing them we are making our programs more flexible and adaptable to meet changing requirements.

Activity 3

Still looking at the String class in the API documentation find other methods that are overloaded.

Feedback 3

There are a few methods that are not overloaded but most are. These include :- Compare(), CompareTo(), Concat(), Join(), Split() and many others.

Looking at the different Substring() methods we see that we can find a substring by either specifying the starting point alone or by specifying starting point and length.

When we use the Substring() method the CLR engine will select the correct implementation of this method, at run time, depending upon whether or not we have provided one or two parameters.

5.3 Summary

Method overloading is the name given to the concept that several methods may exist that essentially perform the same operation and thus have the same name.

The CLR engine distinguishes these by looking at the parameter list. If two or more methods have the same name then their parameter list must be different.

At run time each method call, which may be ambiguous, is resolved by the CLR engine by looking at the parameters passed and matching the data types with the method signatures defined in the class.

By overloading constructors and ordinary methods we are providing extra flexibility to the programmers who may use our classes in the future. Even if these are not all used initially, providing these can help make the program more flexible to meet changing user requirements.

6 Object Oriented Software Analysis and Design

Introduction

This chapter will teach rudimentary analysis and modelling skills through practical examples, leading the reader to an understanding of how to get from a preliminary specification to an Object Oriented Architecture.

Objectives

By the end of this chapter you will be able to….

- Analyse a requirements description
- Identify items outside scope of system
- Identify candidate classes, attributes and methods
- Document the resulting Object Oriented Architecture

This chapter consists of twelve sections :-

1) Requirements Analysis
2) The Problem
3) Listing Nouns and Verbs
4) Identifying Things Outside The Scope of The System
5) Identifying Synonyms
6) Identifying Potential Classes
7) Identifying Potential Attributes
8) Identifying Potential Methods
9) Identifying Common Characteristics
10) Refining Our Design using CRC Cards
11) Elaborating Classes
12) Summary

6.1 Requirements Analysis

The development of any computer program starts by identifying a need :-

- An engineer who specialises in designing bridges may need some software to create three dimensional models of the designs so people can visualise the finished bridge long before it is actually built.

- A manager may need a piece of software to keep track of personnel, what projects they are assigned to, what skills they have and what skills need to be developed etc.

But how do we get from a 'need' for some software to an object oriented software design that will meet this need?

Some software engineers specialise in the task of Requirement Analysis which is the task of clarifying exactly what is required of the software. Often this is done by iteratively performing the following tasks :-

1) interviewing clients and potential users of the system to find out what they say about the system needed
2) documenting the results of these conversations,
3) identifying the essential features of the required system
4) producing preliminary designs (and possibly prototypes of the system)
5) evaluating these initial plans with the client and potential users
6) repeating the steps above until a finished design has evolved.

Performing requirements analysis is a specialised skill that is outside the scope of this text but here we will focus on steps three and four above ie. given a description of a system how do we convert this into a potential OO design.

While we can hope to develop preliminary design skills experience is a significant factor in this task. Producing simple and elegant designs is important if we want the software to work well and be easy to develop however identifying good designs from weaker designs is not simple and experience is a key factor.

A novice chess player may know all the rules but it takes experience to learn how to choose good moves from bad moves and experience is essential to becoming a skilled player. Similarly experience is essential to becoming skilled at performing user requirements analysis and in producing good designs.

Here we will attempt to develop rudimentary skills in the hope that you will have the opportunity to practise those skills and gain experience later.

Starting with a problem specification we will work through the following steps :-

- Listing Nouns and Verbs
- Identifying Things Outside The Scope of The System
- Identifying Synonyms
- Identifying Potential Classes
- Identifying Potential Attributes
- Identifying Potential Methods
- Identifying Common Characteristics
- Refining Our Design using CRC Cards
- Elaborating Classes

By doing this we will be able to take a general description of a problem and generate a feasible, and hopefully elegant, OO design for a system to meet these needs.

6.2 The Problem

The problem for which we will design a solution is 'To develop a small management system for an athletic club organising a marathon.'

For the purpose of this exercise we will assume preliminary requirements analysis has been performed and by interviewing the club managers, and the workers who would use the system, the following textual description has been generated.

The 'GetFit' Athletic Club are organizing their first international marathon in the spring of next year. A field comprising both world-ranking professionals and charity fund-raising amateurs (some in fancy dress!) will compete on the 26.2 mile route around an attractive coastal location. As part of the software system which will track runners and announce the results and sponsorship donations, a model is required which represents the key characteristics of the runners (this will be just part of the finished system).

Each runner in the marathon has a number. A runner is described as e.g. "Runner 42" where 42 is their number. They finish the race at a specified time recorded in hours, minutes and seconds. Their result status can be checked and will be displayed as either "Not finished" or "Finished in hh:mm:ss".

Every competitor is either a professional runner or an amateur runner.

Further to the above, a professional additionally has a world ranking and is described as e.g. "Runner 174 (Ranking 17)".

All amateurs are fundraising for a charity so each additionally has a sponsorship form. When an amateur finishes the race they print a collection list from their sponsorship form.

A sponsorship form has the number of sponsors, a list of the sponsors, and a list of amounts sponsored. A sponsor and amount can be added, and a list can be printed showing the sponsors and sponsorship amounts and the total raised.

A fancy dress runner is a kind of amateur (with sponsorship etc.) who also has a costume, and is described as e.g. "Runner 316 (Yellow Duck)".

6.3 Listing Nouns and Verbs

The first step in analysing the description above is to identify the nouns and verbs:-

- The nouns indicate entities, or objects, some of these will appear as classes in the final system and some will appear as attributes.

- The verbs indicate actions to be performed some of these will appear in the final system as methods.

Nouns and verbs that are plurals are listed in their singular form (e.g. 'books' becomes 'book') and noun and verb phrases are used where the nouns\verb alone are not descriptive enough e.g. the verb 'print' is not as clear as 'print receipt'.

Activity 1

Look at the description above list five nouns and five verbs (use noun and verb phrases where appropriate).

Feedback 1

The list below is a fairly comprehensive list of the nouns and verbs, not just the first five.

Nouns :- GetFit Althletic Club, field, world ranking professional, fund-raising amateur, fancy dress, 26.2 mile route, coastal location, software system, runner, result, sponsorship donation, model, key characteristic, a number, time, result status, competitor, professional runner, amateur runner, world ranking, charity, sponsorship form, collection list, sponsor, sponsorship amount, total raised, costume.

Verbs :- Organise, marathon, compete, track runners, announce results, describe (runner), finish race, specify time, check status, display status, describe (professional), print collection list, add (sponsor and amount), print list, describe (fancy dress runner)

6.4 Identifying Things Outside The Scope of The System

An important part in designing a system is to identify those aspects of the problem that are not relevant or outside the scope of the system. Parts of the description may be purely contextual i.e. for general information purposes and thus not something that will directly describe aspects of the system we are designing. Furthermore while parts of the description may refer to tasks that are performed by users of the system as they are using the system, and thus describe functions that need to be implemented within the system, other parts may describe tasks performed by users while not using the system – and thus don't describe functions within the system.

By identifying things in the description that are not relevant to the system we are developing we keep the problem as simple as possible.

Activity 2

Look at the list of nouns and verbs above and identify one of each that is outside the scope of the system.

Feedback 2

Most of the first paragraph is contextual and does not describe functionality we need to implement within the system. We also need to look at other parts of the description to identify parts that are not relevant.

Things outside the scope of the system…

Nouns :-

- GetFit Althletic Club – this is the client for whom the system is being developed. It is not an entity we need to model within the system.
- Coastal location – the location of the run is not relevant to the functionality of the system as described. Again we do not need to model this as an object within the system.
- Software system – this is the system we are developing as a whole it does not describe an entity within the system.

Verbs :-

- Organise – this is an activity done by members of the athletic club, these may be users of the system but this is not an activity that they are using the system for.
- Marathon – this is what the runners are doing. It is not something the system needs to do.

Note : 'Finish race' is something that a runner does however when this happens their finish time must be recorded in the system. Therefore this in NOT in fact outside the scope of the system.

6.5 Identifying Synonyms

Synonyms are two words that have these same meaning. It is important to identify these in the description of the system. Failure to do so will mean that one entity will be modelled twice which will lead to duplication and confusion.

Activity 3

Look at the list of nouns and verbs and identify two synonyms, one from the list of nouns and one from the verbs.

Feedback 3

Synonyms

Nouns :-

- world ranking professional=professional runner
- fund-raising amateur=amateur runner
- runner=competitor

Note runner is not a synonym of professional runner as some runners are amateurs.

Verbs :-

- marathon=compete
- check status=display status
- print collection list = print list
- finish race = record specified time

6.6 Identifying Potential Classes

Having simplified the problem by identifying aspects that are outside the scope of the system and by identifying different parts of the description that are in reality describing the same entities and operations we can now start to identify potential classes in the system to be implemented.

Some nouns will indicate classes to be implemented and some will indicate attributes of classes.

Good OO design suggests that data and operations should be packaged together – thus classes represent complex conceptual entities for which we can identify associated data and operations (or methods).

Simple entities, such as an address, have associated data but no operations and thus these can be stored as simple attributes within a related class.

Activity 4

Look at the list of nouns above and identify five that could become classes.

Feedback 4

Nouns that could indicate classes:-

- Runner (or Competitor)
- Amateur (or Amateur Runner)
- Professional (or similar)
- FancyDresser (or FancyDressAmateur or similar)
- Sponsorshipform

6.7 Identifying Potential Attributes

Having identified potential classes the other nouns could be used to identify attributes of those classes.

Activity 5

Look at the list of nouns and identify one that could become an attribute of the class 'Runner' and one for the class 'FancyDresser'.

Feedback 5

Nouns that could become attributes...

For Runner :-

number,
resultStatus ie. finished (boolean)
time (hours, minutes, seconds)

For FancyDresser:-
costume (String)

Of course we need to identify all of the attributes for all of the classes.

6.8 Identifying Potential Methods

Having identified potential classes we can now use the verbs to identify methods of those classes.

Activity 6

Look at the list of verbs and identify one that could become a method of the class 'Runner' and one for the class 'FancyDresser'.

Feedback 6

Verbs that could become methods....

For Runner :-

describe (this will actually become an overridden version of ToString())
finishRace
displayStatus

For FancyDresser:-

describe (ToString() will need to be overridden again to encompass the description of the costume)

Of course we need to identify all of the methods for all of the classes.

6.9 Identifying Common Characteristics

Having identified the candidate classes with associated attributes and methods we can start structuring our classes into appropriate inheritance hierarchies by identifying those classes with common characteristics.

Activity 7

Look at the list of classes below and place four into an appropriate inheritance hierarchy. Identify the one class that would not fit into this hierarchy:-

Runner
Amateur
Professional
FancyDresser
Sponsorshipform

Feedback 7

The most general class i.e. the one at the top of the inheritance tree is 'Runner'. Amateur and Professional are subclasses of 'Runner' and FancyDresser is a specific type of Amateur hence a subclass of Amateur.

We can fit these into an inheritance hierarchy because these classes are all related by an is-a relationship. A FancyDresser is-a Amateur which in turn is-a Runner. A Professional is-a Runner as well.

A SponsorshipForm is not a type of Runner and hence does not fit into this hierarchy. This class will be related to one of the other classes by some form of an association. Looking at the description we can see that not all runners have a sponsorship form only amateurs who are running for charity. There is therefore an association between Amateur and SponsoshipFrom. Of course FancyDressers inherit the attributes defined in Amateur and hence they automatically have a SponsorshipFrom.

6.10 Refining Our Design using CRC Cards

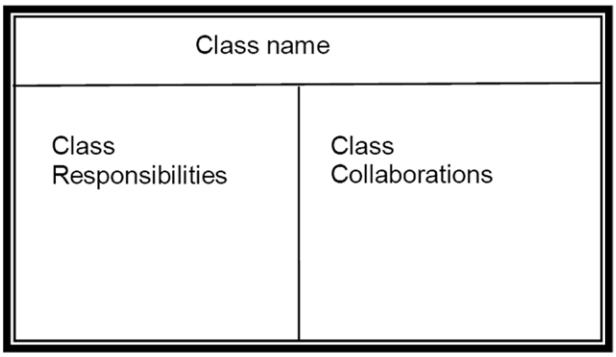
Having identified the main classes in our system, and the attributes and methods of these classes, we could now proceed to refine these designs by defining the data types and other small details, document this information on a UML diagram and program the system. However in a real world system the problem would be larger and less well-defined than the problem we are working on here and the analysis and refinement of design would therefore be a longer more complex process that we can realistically simulate.

As real world problems are more complex our initial designs are unlikely to be perfect therefore it makes sense to check our designs and to resolve any potential problems before turning these designs into a finished system.

One method of doing checking our designs is to document our designs using CRC cards and to check if these work by role-playing different scenarios.

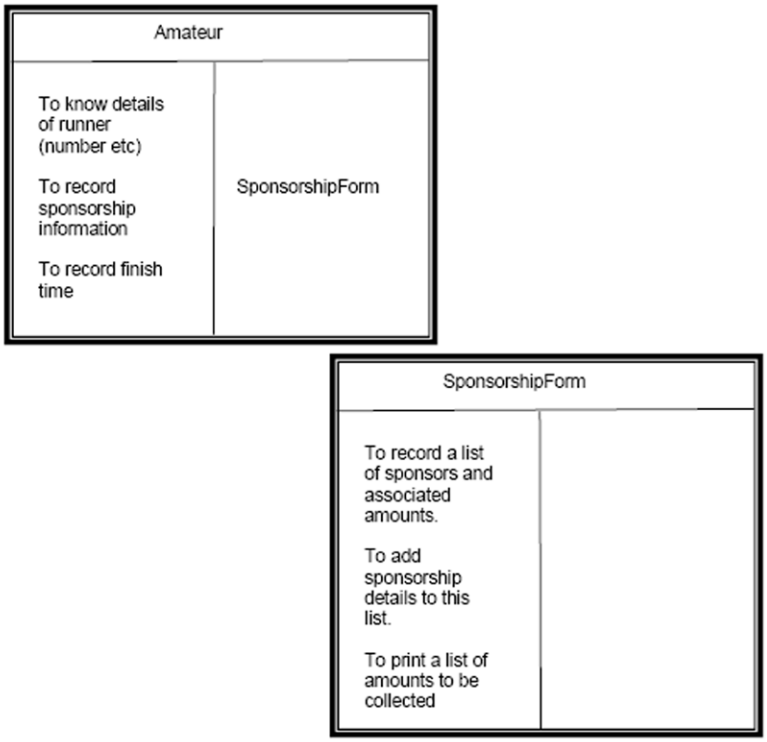
CRC cards are not the only way of doing this and are not part of the UML specification.

CRC stands for Class, Responsibilities and Collaborations. A CRC card is set out below and is made up of three panes with the class responsibilities shown on the left and the collaborations shown on the right.



Responsibilities are the things the class needs to know about (ie the attributes) and the things it should do (ie. the methods) though on a CRC card these are not as precisely defined as on a UML diagram. The collaborations are other classes that this class must work with in order to fulfil its responsibilities.

The diagram below shows CRC cards developed for two classes in the system.



Having developed CRC cards we can role-play a range of scenarios in order to check the system works 'on paper'. If not we can amend the design before getting into the time consuming process of programming a flawed plan.

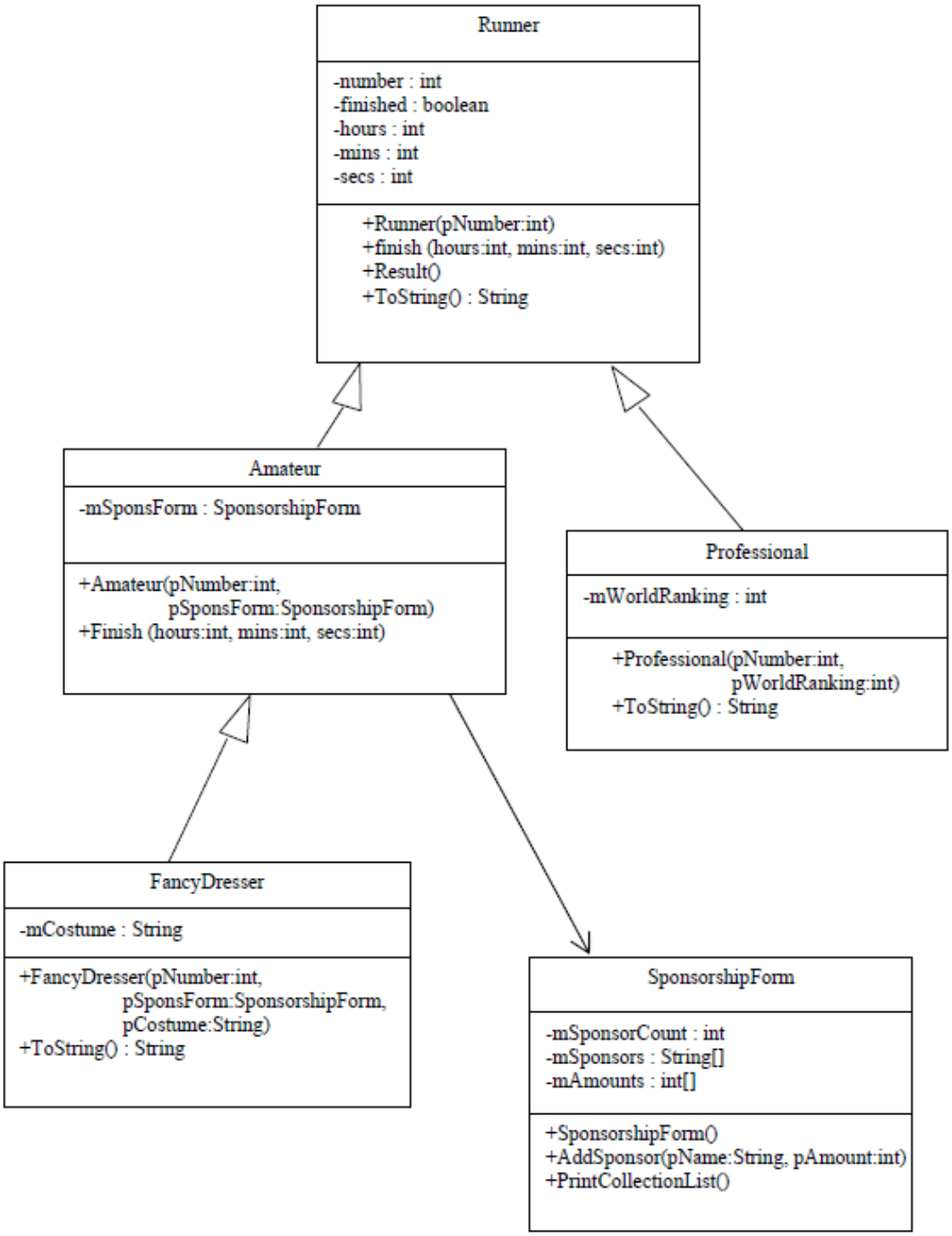
One sample scenario would be when a runner gets an additional sponsor. In this case by looking at the CRC cards above a Runner is able to record sponsorship information in collaboration with the SponsorshipForm class. The SponsorshipForm class records a list of sponsors and can add additional sponsor to this list. So this part of the design seems to work.

Testing out a range of scenarios may highlight flaws in our system designs that we can then fix – long before any time has been wasted by programming weak designs.

6.11 Elaborating Classes

Having identified the classes in our system design, and documented and tested these using CRC cards, we can now elaborate our CRC cards and document our classes using a UML class diagram. To do this we need to take our general specification documented via CRC cards and our resolve any ambiguities e.g. exact data types.

Having elaborated our CRC cards we can now draw a class diagram for proposed design (see below) :-



6.12 Summary

Gathering User Requirements is an essential stage of the software engineering process (and outside the scope of this text).

Turning a complex requirements specification into an elegant simple object oriented architectural design is a skilled task that requires experience. However a good starting point is to follow a simple process set out in this chapter.

Through a sequence of tasks we have seen how to analyse a textual description of a problem. We have :-

- Looked for aspects of the description that are outside the scope of the system,
- Identified where the description refers synonymous items using different words
- Used the nouns and verbs to identify potential classes, attributes and methods
- Looked at the classes to identify potential inheritance hierarchies and to identify other relationships between classes (e.g. associations).
- Document the resulting classes using CRC cards and tested the validity of our design by role-playing a range of scenarios and amending our designs as appropriate
- Finally we can elaborate these details and document the results using a class diagram.

The design process set out in this chapter will be demonstrated again in detail using the case study described in chapter 11.

7 Generic Collections and how to Serialize them

Introduction

This chapter will start by introducing the reader to generic methods it will then go on to introduce the reader to an essential part of the .NET framework :- the classes that implement generic collections. Finally it will introduce the idea of serialization and show how different collections can be serialised as this is a very common task.

Objectives

By the end of this chapter you will be able to….

- Understand the concept of Generic Methods
- Understand the concepts of Collections and the different types of Collection
- Understand the concept of Serialiazation and understand how to serialise the different collections.

This chapter consists of twelve sections :-

1) An Introduction to Generic Methods
2) An Introduction to Collections
3) Different Types of Collections
4) Lists
5) HashSets
6) Dictionaries
7) A Simple List Example
8) A More Realistic Example Using Lists
9) An Example Using Sets
10) An example Using Dictionaries
11) Serializing and De-serializing Collections
12) Summary

7.1 An Introduction to Generic Methods

We have seen previously how methods are identified at run time by their signature i.e. the name of the method and the list of parameters the method takes.

Thus we can have two methods with the same name. Shown below are two methods that find a highest value… one finds the highest value given two integer numbers, the other finds the highest value of two double numbers.

```
static public int Highest(int o1, int o2)
{
     if (o1 > o2)
          return o1;
     else
          return o2;
}
static public double Highest(double o1, double o2)
{
     if (o1 > o2)
          return o1;
     else
          return o2;
}
```

Given the following code the CLR engine will invoke the first of these methods then the second as the correct method to implement is identified by its name and the type of its parameters.

```
int n1 = 1;
int n2 = 2;
Console.WriteLine("The highest is " + Highest(n1, n2));

double n3 = 1.1;
double n4 = 2.2;
Console.WriteLine("The highest is " + Highest(n3, n4));
```

Given the following definition of a Student class…

```
class Student
{
     String name;
     public Student(String name)
     {
          this.name = name;
     }
     override public string ToString()
     {
          return name;
     }
}
```

We could even define a version of this method to find the highest student (alphabetically).

```
static public Student Highest(Student o1, Student o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

And invoke this using ..

```
Student s1 = new Student("Alan");
Student s2 = new Student("Clare");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

In doing this we have created three methods that essentially do exactly the same thing only using different parameter types. This leads us to the idea that it would be nice to create just one method that would work with objects of any type.

The method below is a first attempt to do this :-

```
static public Object Highest(Object o1, Object o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

This method takes any two 'Objects' as parameters. 'Object' is the base class of all other classes thus this method can take any two objects of any more specific type and treat these as of the base type 'Object'. This is polymorphism.

The method above then converts these two 'Object's to strings and compares these strings.

Thus this one method can be invoked three times using the following code….

```
int n1 = 1;
int n2 = 2;
Console.WriteLine("The highest is " + Highest(n1, n2));

double n3 = 1.1;
double n4 = 2.2;
Console.WriteLine("The highest is " + Highest(n3, n4));

Student s1 = new Student("Student A");
Student s2 = new Student("Student B");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

In these cases respectively the CLR engine will treat int, double, and Student objects as the most general type of thing available 'Object'. It will then convert this object to a string and compare the strings.

This will work however in many situations creating methods which take parameters of type 'Object' is flawed or at least very limited.

Inside these methods we do not know what type of object was actually passed as a parameter and hence in the example above we do not know what type of object is actually being returned. When two students object are passed the object returned is a student but we cannot invoke Student specific methods on this object unless we first cast the object returned to a Student.

Assume that we want to invoke an 'AwardMerit() method on the 'Student' returned via the Highest() method. We can do this is we first cast the returned 'Object' onto a 'Student' object. E.g.

```
(Student)Highest(studentA,studentB).AwardMerit();
```

However the compiler cannot be certain that the returned object is a 'Student' and the compiler cannot detect the potentially critical error that would occur is we invoked Highest() on two integer numbers and then tried to cast the returning integer onto an object of type 'Student'.

This leads us to the idea that we would like to be able to create a method that will take parameters of ANY type and return values that are again of ANY type but where we will define these types when we invoke these methods. Such methods do exist and they are called Generic Methods.

Generic methods are methods where the parameter types are not defined until the method is invoked. Parameter types are specified each time the method is invoked and the compiler can thus still check the code is valid.

In other words a generic method uses a parameterized type – a data type that is determined by a parameter.

In the code below a generic method Highest() has been defined as a method that takes two objects as parameters of unspecified type :-

```
static public T Highest<T>(T o1, T o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

We can use this method and each time we invoke it we define the type of object being compared. If two students are compared the compiler will know that the object being returned is also type student and will therefore know it is legal to invoke student specific methods on this object.

A list of generic data types contained between angle brackets that follow the method's name is provided. If multiple generic types are used by a method, their names are separated by commas.

In the example above, the identifer T can stand for any data type. So, when T is used within the brackets in the method's parameter list to describe data, it might be an int, double, 'Student' or any other data type. The only requirement is that the method must work no matter what type of object is passed to it. All objects have a ToString() method because every data type inherits the ToString() method form the 'Object' class, or contains its own overriding version.

In the case above only one generic data type is specified. This is used to define the type of both parameters and the return value.

The generic data type identifier for a generic method can be any legal C# identifier, but by convention it is usually an upper case letter. “T” is used to stand for “type”.

When a generic method is defined i.e. one that works with any data type the type is specified when the method is used. Thus in the code below, while Highest() is a generic method, the compiler can see that a ‘Student’ object is being passed as a parameter and thus the object being returned must also be of type ‘Student’.

```
Student s1 = new Student(“Student A”);
Student s2 = new Student(“Student B”);
Student highestStudent = Highest(s1, s2));
```

Given the object being returned is of type ‘Student’ it must be OK to store this in a variable of type Student and it must be OK to invoke and Student methods on the object returned.

Generic methods can therefore that work with any type data but the compiler can still check for type errors (as the type is specified each time the method is invoked).

Generic classes have been created, based on the same mechanisms as generic methods and these are particularly useful because there were used by the creators of the .NET libraries to create generic collections.

While we won’t be making much use of generic methods and generic classes ourselves in this book we will be making significant use of generic collections, but before using these we need a basic understanding of collections themselves.

7.2 An Introduction to Collections

Most software systems need to store various groups of entities rather than just individual items. Arrays provide one means of doing this, but in C# collections are much more varied and flexible forms of grouping.

In C# collections are classes which serve to hold together groups of other objects. The .NET libraries provides several important classes defined in the System.Collections namespace that provide several different type of collection and associated functionality. This work has been developed and significantly improved upon by the creation of generic collections as defined in the System.Collections.Generic namespace. As generic collections are far more useful than non-generic collections the use of non-generic collections is normally discouraged and this book will focus entirely on generic collections.

7.3 Different Types of Collections

There are three basic type of collection (List, HashSet and Dictionary) defined in the collections namespace.

Activity 1

Go online to msdn.microsoft.com. This is the website for the Microsoft Developer Network and the contains details of the Application Programmer Interface (API) for the .NET framework (it also contains much more).

Follow the links for
Library (tab near top of page)
.NET Development (on left pane)
.NET Framework 4
.NET Framework Class Library

At this point you will see all of the namespaces (packages) in the .NET framework on the left with a description of each on the right)

Follow the link for
System.Collections and then
System.Collections.Generic

Look at the class documentation in the main pane and identify any other collections that you may find useful.

Feedback 1

One of the classes you may have identified is SortedDictionary … this is just like a dictionary that we will cover but one where the elements are automatically sorted for us. We will use a sorted dictionary in the large case study (Chapter 11).

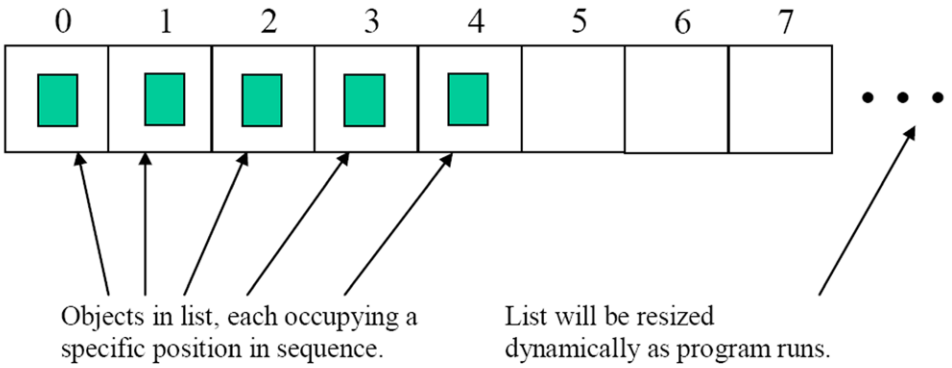
Other useful collections include LinkedList, Queue, and Stack. Discussions of these are beyond the scope of this text but if you have an understanding of Queues, or Linked Lists then it is worth knowing that these have been implemented for you as part of the .NET framework.

We will look at the List, HashSet and Dictionary classes, discuss what they do and show some of the more common useful methods that exist.

7.4 Lists

Lists are the most commonly used type of collection – where you might have used an array, a List will often provide a more convenient method of handling the data.

Lists are very general-purpose data structures, with each item occupying a specific position. They are in many ways like arrays, but are more flexible as they are automatically resized as data is added. They are also much easier to manipulate than arrays as many useful methods have been created that do the bulk of the work for you. Lists store items in a particular sequence (though not necessarily sorted into any meaningful order) and duplicate items are permitted.

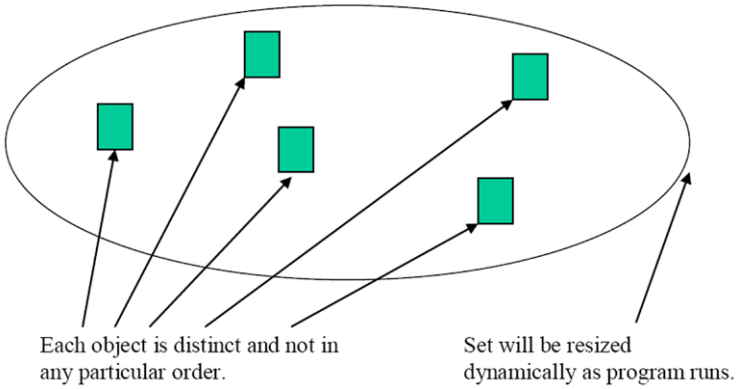


7.5 HashSets

A HashSet is one implementation of a set. A set is like a 'bag' of objects rather than a list. They are based on the mathematical idea of a 'set' – a grouping of 'members' that can contain zero, one or many distinct items.

Unlike a List, duplicate items are not permitted in a set and a Set does not arrange items in order.

Like lists, sets are resized automatically as items are added



Many of the operations available for a List are also available for a Set, but

- we CANNOT add an object at a specific position since the elements are not in any order
- we CANNOT 'replace' an item for the same reason (though we can add one and delete another)
- retrieving all the items is possible but the sequence is indeterminate
- it is meaningless to find what position an element is in.

7.6 Dictionaries

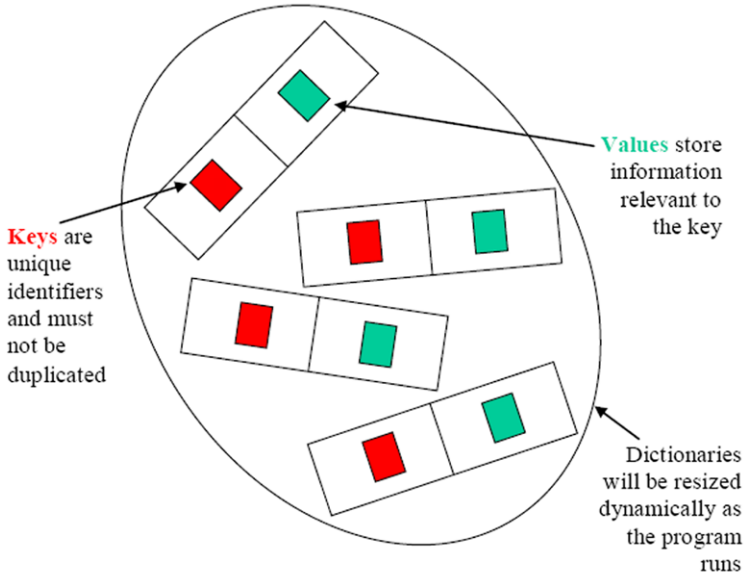
Dictionaries are rather different from Lists and Sets because instead of storing individual objects they store pairs of objects. The pair is made up of a 'key' and a 'value'. The key is something which identifies the pair. The value is a piece of information or an object associated with the key.

In language dictionaries that we are all familiar with the 'key' is the word you look up and the 'value' the meaning of that word. Of course in dictionaries that we are familiar with the entries are sorted into work (or key) order. If that was not the case it would be very difficult to find a word in a dictionary.

Another example would be an address book: in an address book the keys would be people's names and the values their address, phone, email etc. There is only one value for each key, but since values are objects they can contain several pieces of data.

Duplicate keys are not permitted, though duplicate values are. So in the previous example if you looked up two people in the address book you may find them living at the same address – but one person would not have two homes.

Like a set, a Dictionary does not arrange items in order (but a SortedDictionary does, in order of keys) and like lists and sets, dictionaries are resized automatically as items are added or deleted.



Activity 2

For each of the following problems write down which would be most appropriate :- a list, a set or a dictionary.

1) We want to record the members of a club.
2) We want to keep a record of the money we spend (and what it was spent on).
3) We want to record bank account details – each identified by a bank account number

Feedback 2

1) For this we would use a Set. Members can be added and removed as required and the members are in no particular order.
2) For this we would use a List – this would record the items bought in the order in which they were purchased. Importantly as lists allows duplicate items we could buy two identical toys (perhaps for birthday presents for two different children),
3) We would not have two identical Bank accounts so a Set seems appropriate however as each is identified by a unique account number a Dictionary would be the most appropriate choice.

7.7 A Simple List Example

In this section of the book we will demonstrate a simple use of the List collection class.

Activity 3

Go online to msdn.microsoft.com. Find API for the List class defined in the System.Collections.Generic namespace.

List three of the methods you think would be most useful.

Feedback 3

Some of the clearly useful methods include…

Add() – which adds an object onto the end of the list,
Clear() – which removes all the elements from the list,
Contains() – which returns true if this list contains the specified object,
Insert() – which inserts an element at the specified position,
Remove() - which removes the first occurrence of an object from a list and
Sort() – which sorts the element of the list.

Note some methods are overloaded such as Sort() which can either sort the entire list or sort just part of the list specified by a range.

The code below shows an example of the creation of a list. In this case a list of names i.e. Strings.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListOfNames
{
     class Program
     {

          // create names which will store a list of names
          private static List<String> names;

          static void Main(string[] args)
          {
               names = new List<String>(); // Create empty list

               string name;
               for (int i = 0; i < 3; i++) //Enter three names
               {
                    name = Console.ReadLine();
                    names.Add(name);          // Add each name onto list
               }

               foreach (String n in names) // Itterate through list
                                           displaying each element.
               {
                    Console.WriteLine(n);
               }

               Console.WriteLine("Please enter name to search for");
               String searchname = Console.ReadLine();
               if (names.Contains(searchname)) // Demonstrate Contains
                                                    method works.
               {
                    Console.WriteLine("Name found");
               }
               else
               {
                    Console.WriteLine("Name not found");
               }
               Console.ReadLine();
          }
     }
}
```

The code above is fairly self explanatory however perhaps a few parts are worthy of explanation in particular the creation of the list.

Firstly as we are using System.Collections.Generic we are using a generic List class i.e. this can be a list of any object but the type must be specified as the list is created.

Thus the code segment below creates 'names' a variable that will hold a List of Strings. Note the type is specified within the angled brackets < > after the word 'List'.

```
    private static List<String> names;
```

This was created as a Static variable simply for the reason that we are using this directly from within 'Main' and we won't actually be creating an object of this class.

When we invoke the constructor for the List class we must again specify the type of list being created i.e. a list of strings.

```
        names = new List<String>(); // Create empty list
```

In the remainder of the code we a) iteratively add three names to the list, b) display the list and c) use the Contains() method to search for a specific name.

Each element of the list can be accessed in much the same way as elements of an array are accessed (e.g. Console. WriteLine(names[0]) and C# provides an improved 'for loop' construct that will iterate through the elements of the list giving us access to each element as it does so.

Thus as the code below iterates through the list 'n' is set to each element in turn.

```
foreach (String n in names)
{
    Console.WriteLine(n);
}
```

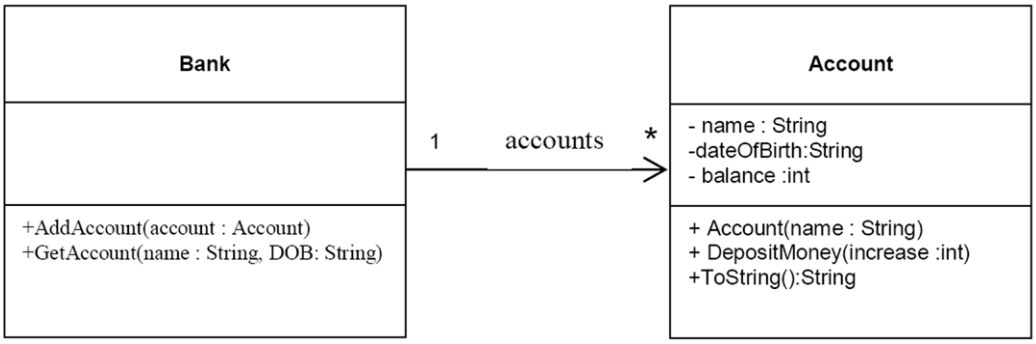
Note elements in a list are stored in order hence when the list is displayed the order remains the same.

While this program stores and manipulates a list of Strings the program could just as easily store a list of Publication – or any object constructed from a class we create.

7.8 A More Realistic Example Using Lists

A more realistic example of lists would come where we are storing more complex objects of our own devising (not just Strings).

Consider the UML class diagram below.



The diagram above represents the basics of a banking system (admittedly a very simplistic banking system). In this system a class is used to simulate a Bank and maintain a collection of Accounts… for the moment we will use a List to manage this collection.

Each element of this list will represent a single account object. One Bank object will maintain a list all accounts.

Each time an account is created the initial balance will be zero but the owner will be able to deposit money when required. So we can focus on the essential purpose of this exercise, to demonstrate the use of a list, we won't bother adding other obviously required functionality (to withdraw money etc).

When each individual account is created we will add it to the list of accounts and we will also allow an individual account to be retrieved so that the account holder can deposit money into their account.

We will demonstrate this with a Main method that runs through a fixed routine a) adding a few accounts, b) displaying these accounts and c) retrieving an account, adding money to this account and displaying the list of accounts again.

We will need an accessor method in Bank so that the Main method can access the list of accounts to display them. In C# we will implement this using a public property.

The code for the Account class is shown below and should need no explanation.

```
class Account
{
     private String name;
     public String Name
     {
          get { return name; }
     }

     private String dateOfBirth;
     public String DateOfBirth
     {
          get { return dateOfBirth; }
     }

     private int balance;

     public Account(String name, String dob)
     {
          this.name = name;
          this.dateOfBirth = dob;
          balance = 0;
     }

     public void DepositMoney(int increase)
     {
          balance = balance + increase;
     }

     public override String ToString()
     {
          return "Name: " + name + "\nBalance : " + balance;
     }
}
```

The code for the Bank class is given below and this demonstrates how easy it is to use the collection class 'List'.

```
class Bank
{
     private List<Account> accounts;
     public List<Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new List<Account>();
     }

     public void AddAccount(Account account)
     {
          accounts.Add(account);
     }

     public Account GetAccount(String name,String dob)
     {
          Account FoundAccount = null;

          foreach (Account a in accounts)
          {
               if ((a.Name == name) && (a.DateOfBirth==dob))
               {
                    FoundAccount = a;
               }
          }
          return FoundAccount;
     }

}
```

Firstly the constructor creates a list of accounts. As with any list this automatically resizes itself, so unlike an array we do not need to worry about running out of space.

Also when elements are removed the remaining elements are moved so that the blank spaces are deleted. While this is not hard to do with arrays it is a time consuming process that is handled automatically for us when we use Lists.

Adding an account becomes a trivial task of invoking the Add() method.

Finally retrieving an individual account is fairly simple to do by iterating through the list until the account we are looking for is found.

Note this list manages a collection of complex objects and when any object is retrieved from the list the methods associated with that object can be invoked on that object.

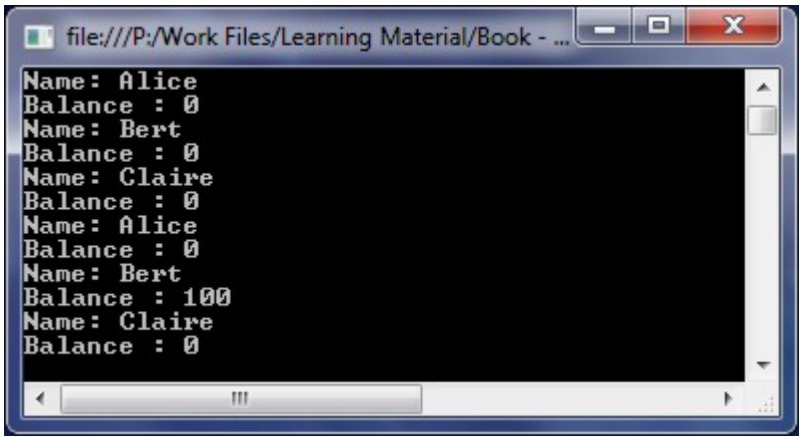
This can be demonstrated via the following 'Main' method.

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;

     a = new Account("Alice","06/12/1963");
     b.AddAccount(a);
     a = new Account("Bert", "14/08/1990");
     b.AddAccount(a);
     a = new Account("Claire","1/1/2000");
     b.AddAccount(a);

     // Display the accounts
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Account a1 = b.GetAccount("Bert", "14/08/1990");
     a1.DepositMoney(100);

     // Display the accounts again
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Console.ReadLine();

}
```

The code above performs the following actions :-

- Firstly it invokes the constructor for the Bank class which in turn creates an empty list of Accounts.
- It then creates three accounts and adds these to the list in the Bank object.
- The list of accounts is then displayed
- An individual bank account is then retrieved from the list and money is deposited into this account.
- Finally the list of accounts is displayed again.

The output from running this program is shown below….

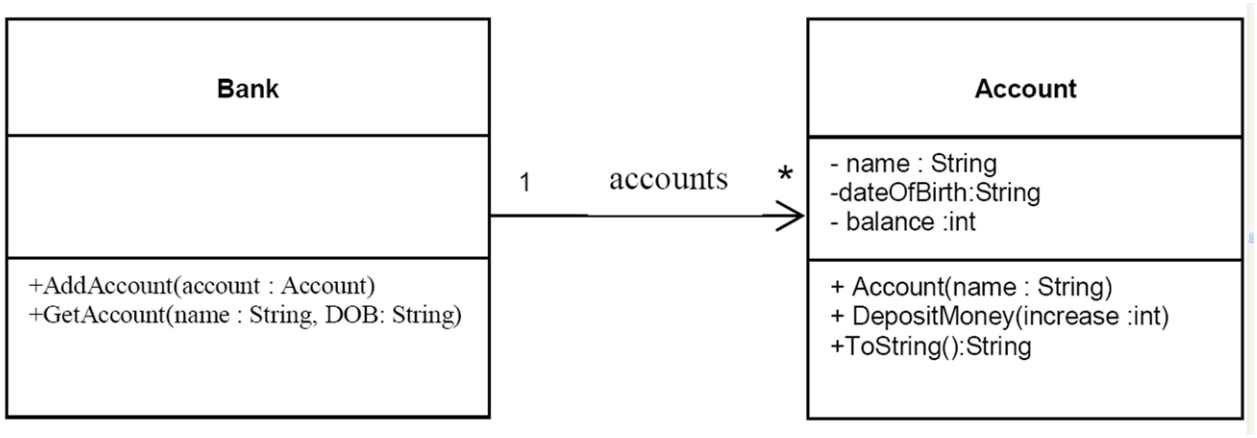


7.9 An Example Using Sets

In the example above we used a list to store a collection of bank accounts. This is not the most sensible choice as allows duplicated to be created and the bank would get very confused if two identical accounts were created (when money was deposited into an account which account should it be added to?).

Furthermore the list stores objects in a particular order – in this case the order the account were created. But this order is irrelevant as it will not help us find a particular account belonging to an individual.

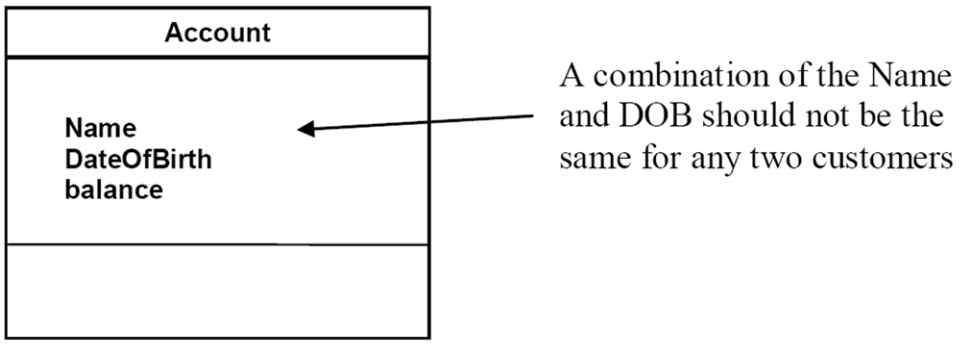
The essential problem remains the same – to store a collection of bank accounts.



However a much better choice for this collection would be to store the collection of accounts as a set, because sets do not allow duplicates to be created.

Consider a set of bank accounts :-

Now consider two bank accounts one for Mr Smith and one for Mrs Jones. To be certain that no two accounts are duplicates we would give each a unique account number – and we will do so later. For now we will assume that no two customers born on the same day will have the same name.



It should not be possible to create a second account where the account holder has the same name and DOB as a previous account holder. However unless told otherwise C# will treat the two objects below as different objects as both objects have different names (Account1 and Account2) and thus while sets do not allow duplicates objects both of these accounts could be created and added to a set because the computer does not recognize them as duplicates..

Account1
Mr Smith 1/1/1990 £1,200

Account2
Mr Smith 1/1/1990 £0

To overcome this problem we need to override the Equals() method defined in the Object class to say these objects are the same if the Name and DOB are the same.

We can do this as below….

```
public override bool Equals(object obj)
{
     Account a = (Account)obj;
     return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
}
```

This overrides Object.Equals() for the Account class

Even though it will always be an Account object passed as a parameter, we have to make the parameter an Object type (and then cast it to Account) in order to override the Equals() method inherited from Object which has the signature

public bool Equals (Object obj)

(You can check this in the by looking for the 'Object' class in the 'System' namespace of the .NET API)

If we gave this method a signature with an Account type parameter it would not override Object.Equals(Object obj). It would in fact overload the method by providing an alternative method. As the system is expecting to use a method with a signature of public bool Equals (Object obj) it would not use a method with a signature of public bool Equals (Account obj).

Therefore we need to override public bool Equals (Object obj) and cast the parameter to an Account before extracting the name and DOB of the account holder to compare them with those of the current object.

One additional complication concerns how objects are stored in sets. To do this C# uses a hash code. Two accounts with the same name and DOB should generate the same hash code however currently this will not be the case as the hash code is generated using the object name (e.g. Account1). Thus two accounts, Account 1 and Account 2 could still be stored in the set even if the name and DOB is the same.

To overcome this problem we need to override the GetHashCode() method so that a hash code is generated using the Name and DOB rather than the object name (just as we needed to override the Equals() method).

We can ensure that the hash code generated is based on the name and DOB by overriding the GetHashCode() method as shown below….

```
public override int GetHashCode()
{
     return (name + dateOfBirth).GetHashCode();
}
```

The simplest way to redefine GetHashCode() for an object is to join together the instance values which are to define equality as a single string and then take the hashcode of that string. In this case equality is defined by the name of the account holder and DOB which taken together we assume will be unique.

It looks a little strange, but we can use the GetHashCode() method on this String even though we are overriding the GetHashCode() method for objects of type Account.

GetHashCode() is guaranteed to produce the same hash code for two Strings that are the **same**. Occasionally the same hash code may be produced for **different** key values, but that is not a problem.

By overriding Equals() and GetHashCode() methods C# will prevent objects with duplicate data (in this case with duplicate name and dates of birth) from being added to the set.

The full code for the class Account is given below.

```
class Account
{
     private String name;
     public String Name
     {
          get { return name; }
     }

     private String dateOfBirth;
     public String DateOfBirth
     {
          get { return dateOfBirth; }
     }

     private int balance;

     public Account(String name, String dob)
     {
          this.name = name;
          this.dateOfBirth = dob;
          balance = 0;
     }

     public void DepositMoney(int increase)
     {
          balance = balance + increase;
     }

     public override String ToString()
     {
          return "Name: " + name + "\nBalance : " + balance;
     }

     public override bool Equals(object obj)
     {
          Account a = (Account)obj;
          return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
     }

     public override int GetHashCode()
     {
          return (name + dateOfBirth).GetHashCode();
     }
}
```

The code above is identical to the previous version of the Account class (when we used lists) except for the addition the overridden methods Equals() and GetHashCode(). When storing any object in a set we must override both of these methods to prevent duplicates being stored.

The code for the Bank class is identical apart from the firs few lines which creates a typed HashSet instead of a typed List.

```
class Bank
{
     private HashSet<Account> accounts;
     public HashSet<Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new HashSet<Account>();
     }

     public void AddAccount(Account account)
     {
          accounts.Add(account);
     }

     public Account GetAccount(String name,String dob)
     {
          Account FoundAccount = null;

          foreach (Account a in accounts)
          {
               if ((a.Name == name) && (a.DateOfBirth==dob))
               {
                     FoundAccount = a;
               }
          }
          return FoundAccount;
     }

}
```

Finally the Main method used to demonstrate this is shown below…

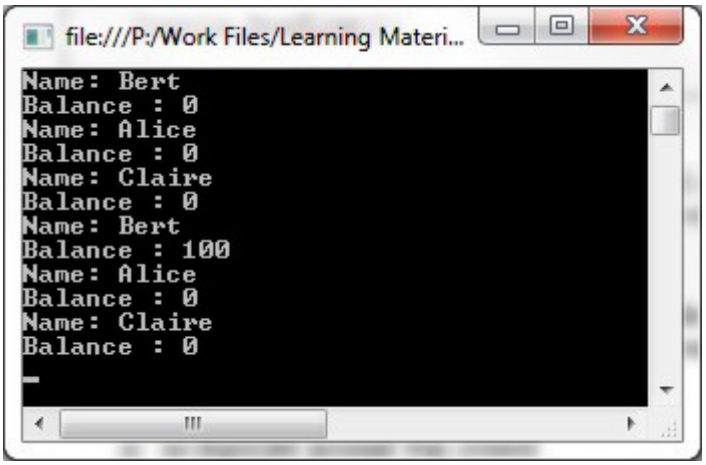
```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(a);

     // try to create and add a duplicate account-this shouldn't work
     a = new Account("Bert", "06/12/1963");
     b.AddAccount(a);

     a = new Account("Alice", "14/08/1990");
     b.AddAccount(a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(a);



     // Display the accounts
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Account a1 = b.GetAccount("Bert", "06/12/1963");
     a1.DepositMoney(100);

     // Display the accounts again
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Console.ReadLine();

}
```

This Main method is very similar to the Main method used to test lists. Several accounts are created, displayed, an account is retrieved and after a deposit into this account has been made the list is displayed again.

Two changes should be noted. First an attempt is made to create a duplicate account – this should not work. The list of accounts has been created in a non alphabetical order.

By looking at the program output, shown below, we can see that :-

a) no duplicate account was created, so overriding Equals() and GetHashCode() worked.
b) the accounts are listed in the order created but not in an meaningful order. Entries in sets are not stored in any order and therefore we cannot be sure which order they will be retrieved in.



Activity 4

Earlier we found that some of the useful methods for Lists included…

Add() – which adds an object onto the end of the list,
Clear() – which removes all the elements from the list,
Contains() – which returns true if this list contains the specified object,
Insert() – which inserts an element at the specified position,
Remove() - which removes the first occurrence of an object from a list and
Sort() – which sorts the element of the list.

Go online to msdn.microsoft.com. Find API for the HashSet class defined in the System.Collections.Generic namespace and find out which of the methods in the List class have equivalent methods in the HashSet class.

Feedback 4

You should find Methods Add(), Clear() Contains() and Remove() methods also exist for a HashSet.

Insert() and Sort() do not exist. As sets are unsorted objects you cannot sort them and as they have no specific order it make no sense to try to insert an object at a specified position.

There is much commonality between the different types of collection. Having learnt how to use one it is relatively easy to learn another.

Activity 5

The code below will find a String in a set of Strings (called StringSet). Amend this so this will work find a Publication in a set of Publications (called PublicationSet).

```
public void FindString(String s)
{
     boolean found;
     found = StringSet.Contains(s);
     if (found)
     {
          Console.WriteLine("Element " + s + " found in set");
     }
     else
     {
          Console.WriteLine("Element " + s + " NOT found in set");
     }
}
```

Feedback 5

```
public void FindPublication(Publication p)
{
     boolean found;
     found = PublicationSet.Contains(p);
     if (found)
     {
          Console.WriteLine("Element " + p + " found in set");
     }
     else
     {
          Console.WriteLine("Element " + p + " NOT found in set");
     }
}
```

Activity 6

Consider the set of Publications that would need to be created for the code in Activity 5 to work and answer the following questions.

1) Could such a set be used to store a collection of books?

2) Could it store a combination of books and magazines?

3) If a book was found in the set what would the following line of code do?

Console.WriteLine("Element " + p + " found in set");

Feedback 6

1) Yes. Books are a subtype of publication and could be stored in a set of publications.

2) Yes. For the same reasons we could store a combination of books and magazines objects (or any other type of publication).

3) 'p' would invoke the ToString() method on the publication. The CLR engine would determine at run time that this was in fact a book and assuming the ToString() method had been overridden for book, to return the title and author, this would make up part of the message displayed. Thus polymorphically the message would change depending upon which type of publication was found in the collection.

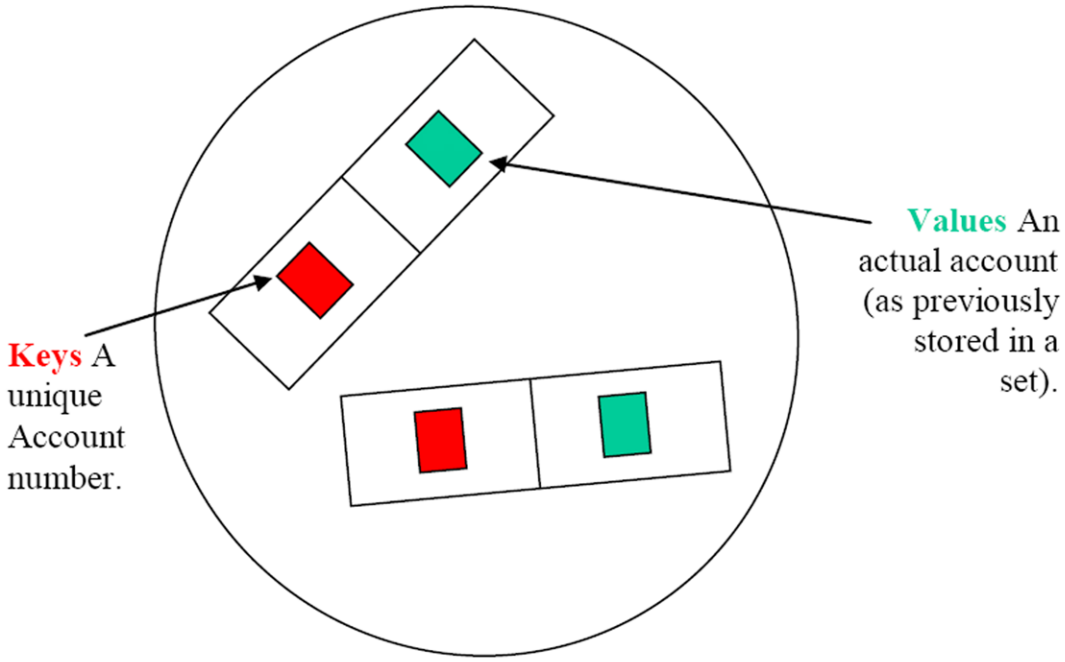
7.10 An Example Using Dictionaries

In the previous example we assumed no bank accounts would be created for two customers with the same name and DOB. However it is possible that two customers could have the same name and DOB.

To make accounts truly unique we could amend the definition of Account to contain an account number that we will ensure is unique and we could continue to store these using a set. However as we will often retrieve account using an account number a dictionary would be a more efficient collection for this application.

A dictionary is like a set but where each item stored is made up of a key/value pair.

In this case the key would be a unique account number and the value would be the associated bank account object.



In the previous example we stored a collection of Accounts in a set, we do not need to make any changes to the Account class if we wish to store these in a dictionary.

We could remove the overridden methods for Equals() and GetHashCode() from the Account class as a dictionary can contain duplicate values. Alternatively we could leave these methods in should we wish to.

In a dictionary it is the keys that are unique.

There are some significant changes to the Bank class that revolve around the fact that it is a Dictionary not a set that is being created. The code for the Bank class is shown below :-

```
class Bank
{
     private Dictionary<int,Account> accounts;
     public Dictionary<int,Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new Dictionary<int, Account>();
     }

     public void AddAccount(int number,Account account)
     {
          try
          {
               accounts.Add(number, account);
          }
          catch (Exception)
          {
          }
     }

     public Account GetAccount(int number)
     {
          Account a;
          accounts.TryGetValue(number, out a);
          return a;
     }
}
```

In the code above when creating the dictionary you can see that two data types are specified :-. Firstly the type for the dictionary key is specified – in our case we will use a simple 'int' for an account number. Secondly the data type for the value is specified – in our case the value is a complete account object. Hence the segment of code below :-

```
     private Dictionary<int,Account> accounts;
```

The dictionary class defines two very useful methods for us, Add() and TryGetValue().

The Add() methods requires two parameters, the 'Key' and the 'Value'. Hence we must provide both the account number and the account object to be added to the dictionary :-

```
          accounts.Add(number, account);
```

Note this method will generate an exception if the 'key' is not unique and this error should be trapped and dealt with ….though in this simple case we have decided to take no action should this occur.

The TryGetValue() method requires an input argument, the key it is searching for and sets the value of the second argument to the value associated with that key. In our example it sets the second parameter to the account associated with the account number. Note this will be null if the account number does not exist. See code below :-

```
accounts.TryGetValue(number, out a);
```

The Main method used to test this dictionary is given below :-

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;
     int an;     // account number;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1,a);

     // try to create and add a duplicate account-this shouldn't work
     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1, a);

     a = new Account("Alice", "14/08/1990");
     b.AddAccount(2, a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(3, a);

     // Display the accounts
     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     a = b.GetAccount(1);
     a.DepositMoney(100);

     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;
          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     Console.ReadLine();
}
```

Functionally the 'Main' method above is virtually identical to the Main method used to test the Set example.

The code above performs the following operations :-

- Several accounts are created, including one attempt to create a duplicate,
- The collection of accounts is displayed, in this case including an account number,
- an account is retrieved and amended and then
- the collection is displayed again.

To keep this example short we have not protected against the possibility that the account being retrieved cannot be found.

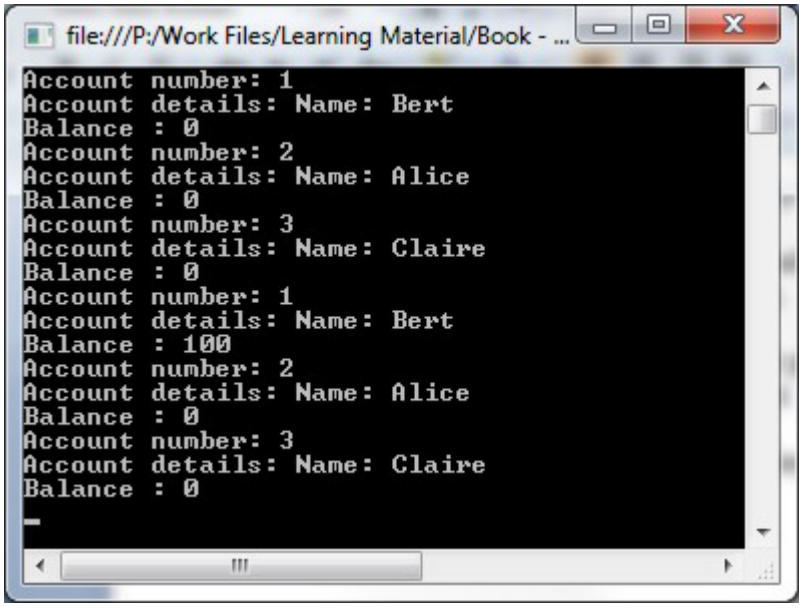
The following lines of code which display the contents of the collection are perhaps worthy of a fuller explanation :-

```
foreach (KeyValuePair<int, Account> kvp in b.Accounts)
{
     an = kvp.Key;
     a = kvp.Value;
          Console.WriteLine("Account number: " + an +"\nAccount details: " + a.ToString());
}
```

Each object in the collection is made up of a key/value pair. Thus when we iterate around the collection each iteration returns a key/value pair.

Above we define kvp a variable made up of a key/value pair where the key is an 'int' i.e. an account number and the value is an Account object We split this pair into its component parts and store these in the respective variables ('an' and 'a'). We then display the account number and invoke the ToString() method on the account, using the result to display details of the account.

The output from this program is shown below:-



As we would expect the system did not allow us to create accounts with duplicate account numbers.

If you look at the Dictionary class in the System.Sollections.Generic namespace of the .NET framework you will see a host of other useful methods that have been created to manage Dictionaries. These include Clear(), ContainsKey(), ContainsValue() and Remove().

7.11 Serializing and De-serializing Collections

Collections, Lists, Sets and Dictionaries among others, are very powerful flexible mechanisms for storing collections of objects.

They automatically resize themselves and contain methods that save significant programming effort when adding members, retrieving members, removing members, searching for members, sorting collections etc.

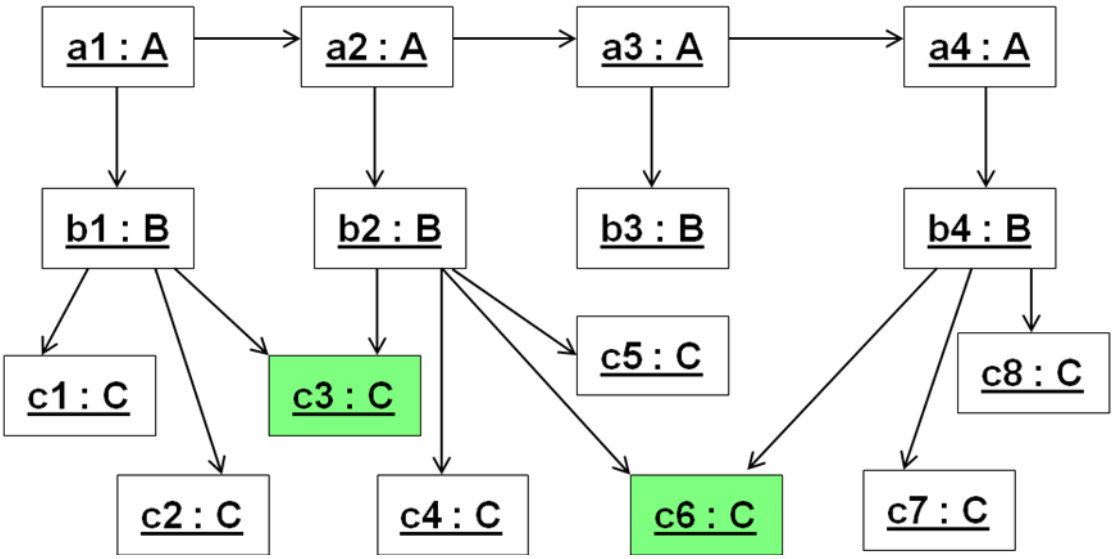
They become even more powerful when combined with the serialization / de-serialization facilities provided by the .NET framework.

Using traditional file handling routines textual data can be stored and retrieved from files.

Serialization allows whole objects to be stored with one simple command (once an appropriate file has been opened).

However C# objects frequently contain references to other objects, they create what is known formally as a 'graph' – a network of connections. The serialization mechanism follows these references and also serializes the objects referenced… and objects those objects reference… etc.

An object graph is shown below :-



In this graph we can see that object a1 refers to objects a2 and object b1. These in turn refer to other objects. Thus if we were to fully save (or serialise) a1 we would need to store all the details of this object including details of all the other objects this refers to… and all the objects these refer to and so on.

Serialisation will follow these links automatically.

Notice that c3 and c6 are referred to by two objects. However, the serialization mechanism is smart enough to realise that the same object is occurring when it encounters it for the second time and instead of writing out a duplicate it simply writes out a reference to the original object. By the same means a 'cycle' in the graph – i.e. a circle or references returning to the same place, will not cause the serialization mechanism to go into an infinite loop!

This mechanism assumes that classes A, B and C are all serializable. Some classes are not. For instance those that rely on reading from a file cannot be serialised as the file may not exist when we attempt to deserialize an object of that class. In practice most of the classes we create will be serializable.

To demonstrate the power of the serialization mechanism we will amend the bank account system developed in section 7.10 where a dictionary was used to store a collection of accounts.

With one instruction we will store an object of type Bank. In doing so the system will also store the collection Accounts and in doing so it will store ALL Account objects.

When we retrieve the Bank object all Account objects will also be retrieved and the collection reconstructed.

In order to do this no changes need to be made to the Account class or the Bank class though we must tell the compiler these classes can be serialiazed.

To do this we place the keyword serializable in front of each class as shown below.…

```
[Serializable]
class Account
{
     // code from body of class omitted
}
[Serializable]
class Bank
{
     // code from body of class omitted
}
```

In the 'Main' method that we will use to invoke the serialization \ de-serialization process, and to test the results, we need to add additional using statements as we are using additional parts of the .NET framework (see below).

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
```

A complete 'Main' method that will demonstrate this working is provided below. Much of this is either self explanatory or contains code we are already familiar with. The new parts that relate to the serialiazation / de-serialization will be explained afterwards.

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;
     int an;     // account number;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1, a);
     a = new Account("Alice", "14/08/1990");
     b.AddAccount(2, a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(3, a);

     Console.WriteLine("Stored data");
     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                              "\nAccount details: " + a.ToString());
     }

     FileStream outFile = new FileStream("AccountData",
                                        FileMode.Create, FileAccess.Write);
     BinaryFormatter bFormatter = new BinaryFormatter();
     bFormatter.Serialize(outFile, b);
     outFile.Close();
     outFile.Dispose();

     Bank b2 = new Bank();
     FileStream inFile = new FileStream("AccountData", FileMode.Open,
                                                          FileAccess.Read);
     b2 = (Bank)bFormatter.Deserialize(inFile);
     inFile.Close();
     inFile.Dispose();

     Console.WriteLine("\nRetrieved data");
     foreach (KeyValuePair<int, Account> kvp in b2.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     Console.ReadLine();

}
```

The code above performs the following steps :-

- Firstly several bank accounts are created, added to the collection and the collection is then displayed.
- Next an output file is created
- A binary formatter is created as this is used by the serialization process.

- With one line the collection 'b' is serialised and in doing so all account objects inside this collection are serialized (see code below).

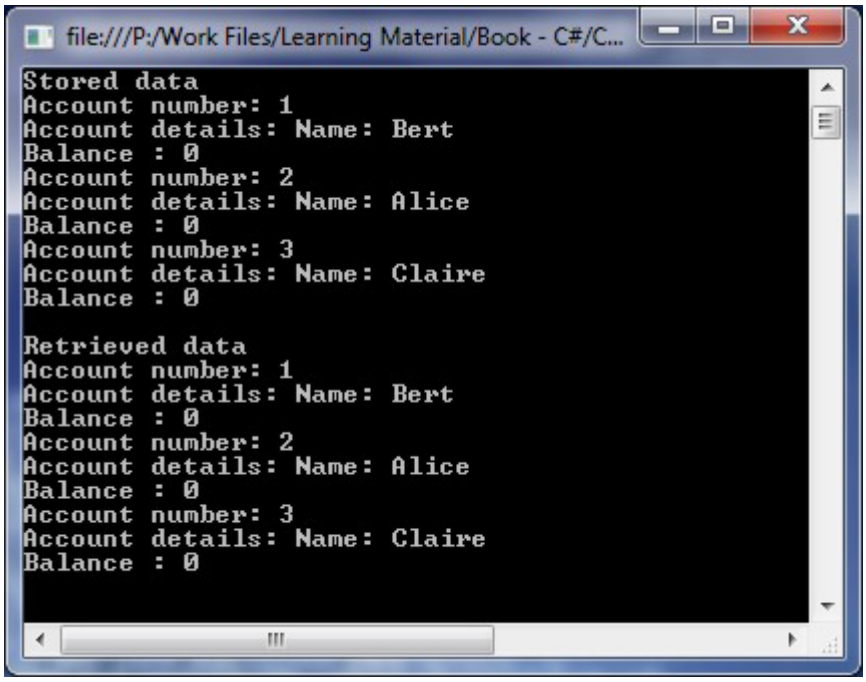
```
bFormatter.Serialize(outFile, b);
```

- In order to prove this has worked a new, empty Bank object is created 'b2'.
- The de-serialization process is invoked and the output cast into a Bank type. The results are then stored in the object 'b2' (see code below).

```
b2 = (Bank)bFormatter.Deserialize(inFile);
```

Finally the contents of 'b2' are displayed to show they are consistent with the data originally created.

The results of running this program are shown below.



7.12 Summary

The .NET Collections classes provide ready-made methods for storing collections of objects. These almost completely make the use of arrays redundant!

There are 'untyped' collections and 'typed' collections. Typed collections, or generic collections, were developed based upon the idea of generic methods and generic classes. The use of generic collections are much more useful than untyped collections and thus we have not shown the use untyped collections here.

Collection classes include List, HashSet and Dictionary. Each of these define appropriate methods, many of which are common across all of the collection classes.

Special attention is required when defining objects to be stored in Sets (or as keys in Dictionaries) to define the meaning of 'duplicate'. For these we need to override the Equals() and GetHashCode() methods inherited from Object.

Serialization is a very powerful mechanism that allows the entire contents of a collection and all related objects to be stored with one simple command.

A further example of the use of collections will be provided in Chapter 11, a larger case study at the end of this book. This will demonstrate the use of lists, sets and dictionaries for a small but more realistic example application. The code for this will be available to download and inspect if required.

8 C# Development Tools

Introduction

This chapter will introduce the reader to several development tools that support the development of large scale C# systems. We will also consider the importance of documentation and show tools can be used to generate documentation for systems you create (almost automatically).

Objectives

By the end of this chapter you will be able to….

- Find details of several professional and free interactive development environments
- Understand the importance of the software documentation tools and the value of embedding XML comments within your code.
- Write XML comments and generate automatic documentation for your programs.

This chapter consists of eight sections :-

1) Tools for Writing C# Programs….
2) Microsoft Visual Studio
3) SharpDevelop
4) Automatic Documentation
5) Sandcastle Help File Builder
6) GhostDoc
7) Adding Namespace Comments
8) Summary

8.1 Tools for Writing C# Programs

Whatever mode of execution is employed (see section 1.7), programmers can work with a variety of tools to create source code. It is possible to write C# programs using simple discrete tools such as a plain text editor (e.g. Notepad) and a separate compiler invoked manually as required. However virtually all programmers would use a powerful Integrated Development Environment (IDE) which use compilers and other standard tools behind a seamless interface.

Even more sophisticated tools Computer Aided Software Engineering (CASE) tools exist which integrate the implementation process with other phases of the software development lifecycle. CASE tools could take UML class diagrams, generated as part of the software analysis and design phase, and generate classes and method stubs automatically saving some of the effort required to write the C# code (ie. the implementation phase.

CASE tools could also help by automating the software testing phase.

Overtime, as more powerful tools are being created, IDEs have begun to incorporate some of the features previously found only in CASE tools i.e. they do much more than allow programs to be written, edited and compiled. One such tool is Microsoft Visual Studio.

8.2 Microsoft Visual Studio

Moving to an 'industrial strength' IDE is an important stepping stone in your progress as a software developer, like riding a bicycle without stabilisers for the first time. With some practice you will soon find it offers lots of helpful and time-saving facilities that you will not want to work without again.

Microsoft Visual Studio is a powerful IDE platform that supports the development of .NET programs written in a range of languages not just C#.

Details of this can be found via http://www.microsoft.com/visualstudio/.

Visual Studio comes in many varieties but perhaps the two most common varieties are Visual Studio Professional edition and Visual Studio Express edition. The professional was used to create all of the code in this book and for the case study application developed in Chapter 11. However the express edition is free and still contains all of the features required to create and run the applications in the book.

For our purposes there is only one thing missing from the Express edition is the unit testing tools explained in section 10.5 and used in section 11.12. In all other respects it is a very powerful IDE and fully supports a student wanting to learn and develop C# programs.

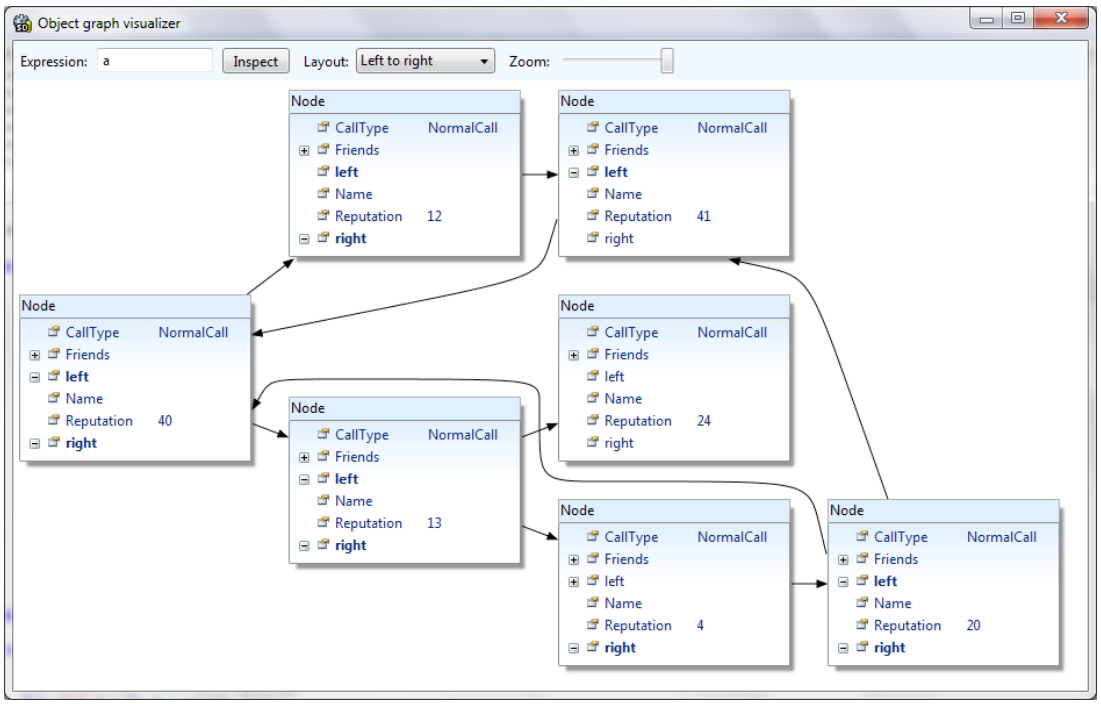
Visual Studio Express edition can be downloaded from the following website http://www.microsoft.com/express/Downloads/

8.3 SharpDevelop

Another very powerful and free tool that can be used to develop C# programs is SharpDevelop (or #develop). It is open-source so you can download both the sourcecode and executable versions.

SharpDevelop includes support for several .NET languages including C# 4.0 and VB.NET 10. It includes unit testing facilities (via optional plug ins), code completion facilities and powerful debugging facilities.

To help debugging, SharpDevelop includes an Object graph visualiser (see below). This visualiser displays a visual representation of your data structures that is dynamically updated when stepping through your code.



For more information on SharpDevelop or to download it go to http://community.sharpdevelop.net/forums/t/12513.aspx

It should be noted that other free tools exist including MonoDevelop (monodevelop.com/) an IDE which offers multiplatform support including Linux, Windows and Mac OSX).

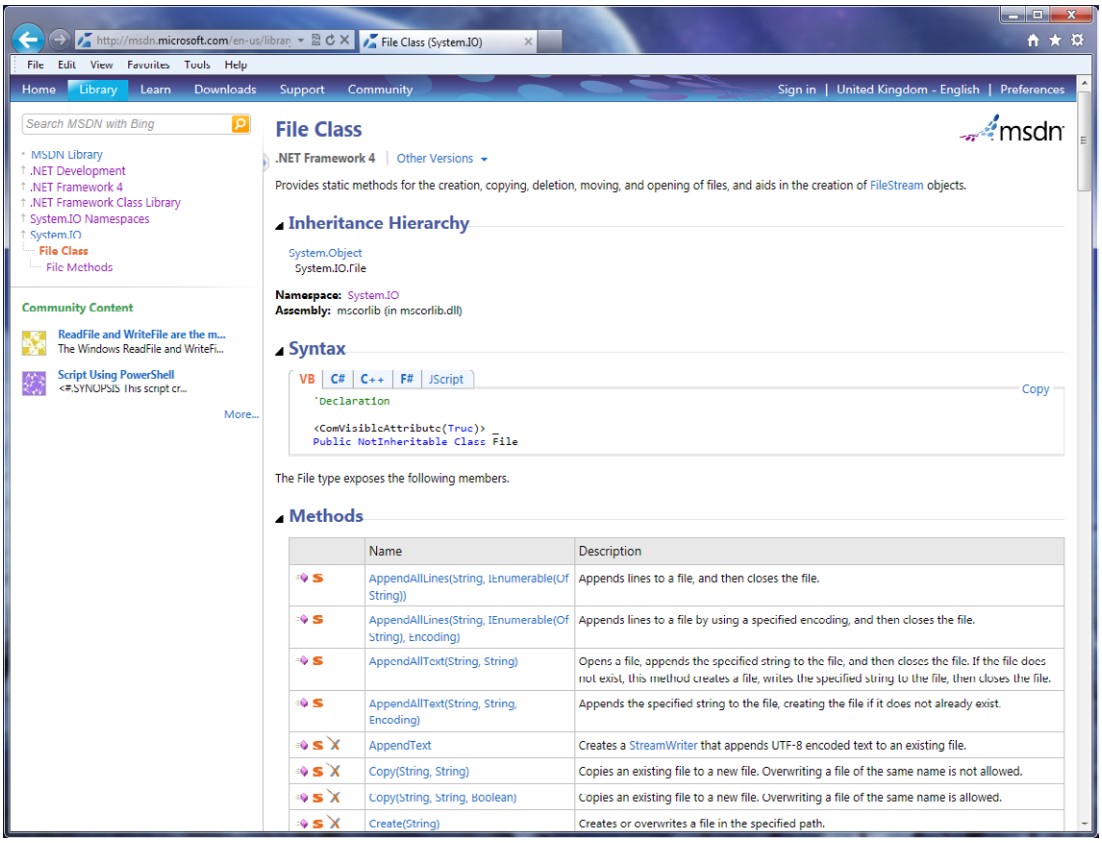
8.4 Automatic Documentation

One particularly useful feature found within some IDE's is the ability to generate automatic documentation for the programs we create.

Programmers for many years have been burdened with the tedious and time consuming task of writing documentation. Some of this documentation is intended for users and explain what a program does and how to use it. Other documentation is intended for future programmers who will need to amend and adapt the program so that its functionality will change as the needs of an organisation change. These programmers need to know what the program does and how it is structured. They need to know :-

- what packages it contains
- what classes exist in each of these packages and what these classes do,
- what methods exist in each class and for each of these methods
    - what does the method do
    - what parameters does it require
    - what, if any, value is returned.

Tools can't produce a user guide but they can provide a technical description of the program. Tools can analyse C# source files for the programs we write and produce documentation, either as a set of web pages (HTML files) or in some other format. Technical documentation should contain similar information as you would find when looking MSDN to find details of the .NET libraries.



This website provides, as a set of indexed web pages, essential details of the .NET libraries including all packages, classes, methods and method parameters and return values. This extremely useful information was not generated by hand but generated using the automatic tools.

Tools can produce similar detailed documentation for any C# program you create and this would help future programmers amend and update your programs. However this relies on properly formatted (and informative!) XML-style comments in source files, including tags such as @author, @param etc.

Because this documentation is generated automatically, at the push of a button, it saves programmers from a tedious, time consuming and error prone task. Furthermore the documentation can be updated whenever a program is changed.

XML (Extensible Markup Language) is a way of specifying and sharing structured information.

By adding meaningful XML comments to the code we write tools can produce automatic documentation for us. Poor attention to commenting of source code will result in poor documentation. On the other hand, if the commenting is done properly then the reference documentation is produced for virtually no effort at all.

XML comments should therefore be added at the start of every class using the standard tags <summary> and <remarks> to give an overview of that class in the following format..

/// <summary>
/// Provide a short description of the lass and its role here ….
/// </summary>
/// <remarks>Author Simon Kendal
/// Version 1.0 (5th May 2011)</remarks>

Similar comments should be provided for every method using the /// <param name=”name of paramter”>, and /// <returns> tags to describe each parameter and to describe the value returned. The details of each parameter, starting with the name of the parameter, should be provided on separate lines as shown below.

/// <summary>
/// A description of the method
/// </summary>
/// <param name=”name of first parameter”>A description of that parameter </param>
/// <param name=”name of 2nd parameter”>A description of that parameter </param>
/// <returns> A description of the value returned by a method. /// </returns>

Activity 1

The method below takes two integer numbers and adds them together. This value is returned by the method. Write an XML comment, using appropriate tags, to describe this method.

```
public int add(int number1, int number2)
{
return (number1 + number2);
}
```

Feedback 1

```
/// <summary>
/// This method adds two integer numbers.
/// </summary>
/// <param name=”number1”>The first number.</param>
/// <param name=”number2”>The second number.</param>
 /// <returns>The sum of the two numbers. </returns>
```

Automatic tools cannot analyse comments to determine if these provide an accurate description of the methods, however tools can do far more than cut and paste a programmers comments onto a web document. One thing tools do is analyse method signatures and compare this with the tags in the comments to check for logical errors. If a method requires three

parameters but the comment only describes two then this error will be detected and reported to the programmer.

```
"Parameter 'name' has no matching param tag in the XML comment for 'Name of class here' (but
other parameters do)".
```

By analysing the code the tools can also provide additional information about the class, the methods that have been inherited and the methods that have been overridden.

By using automatic tools reference documentation can be produced for programmers using the class(es) concerned, it does not include comments within methods intended for programmers editing the class source code in maintenance work.

Automatic documentation tools do not exist as standard within either Microsoft Visual Studio or SharpDevelop. However both of these IDE's can be integrated with Sandcastle Help File Builder… this then enables automatic documentation to be generated. Better still another tool exists called GhostDoc which further helps by generating the necessary XML comments.

8.5 Sandcastle Help File Builder

To be precise Sandcastle Help File Builder (SHFB) does not produce automatic documentation for your code… but it adds a graphical front end to 'Sandcastle'. 'Sandcastle' is a difficult to use tool that lacks a GUI, and has a complex installation routine but will generate automatic documentation.

Sandcastle Help File builder is a GUI that will drive Sandcastle and will also automate the installation of Sandcastle. Sandcastle Help File Builder can be downloaded from http://shfb.codeplex.com/.

Running this software will install

a) Sandcastle,
b) updates patches and additional files
c) compilers for different help file formats – only Help 1 HTML compiler required.
d) Sandcastle Help File Software itself.

When running SHFB you need to specify an XML file with comments generated from your code and the C# code itself i.e. a .sln file

To generate the XML file from within Visual Studio select the project Properties / Build / XML comments. Note Visual Studio will then complain if comments are missing from your code.

8.6 GhostDoc

SHFB is very useful as it generates automatic documentation for your programs assuming of course appropriate XML comments exist in the body of the code.

GhostDoc is one piece of software that will automatically add XML comments to your C# code. Though the comments should be edited to ensure they are meaningful.

GhostDoc can be downloaded from http://submain.com/products/ghostdoc.aspx

8.6 Adding Namespace Comments

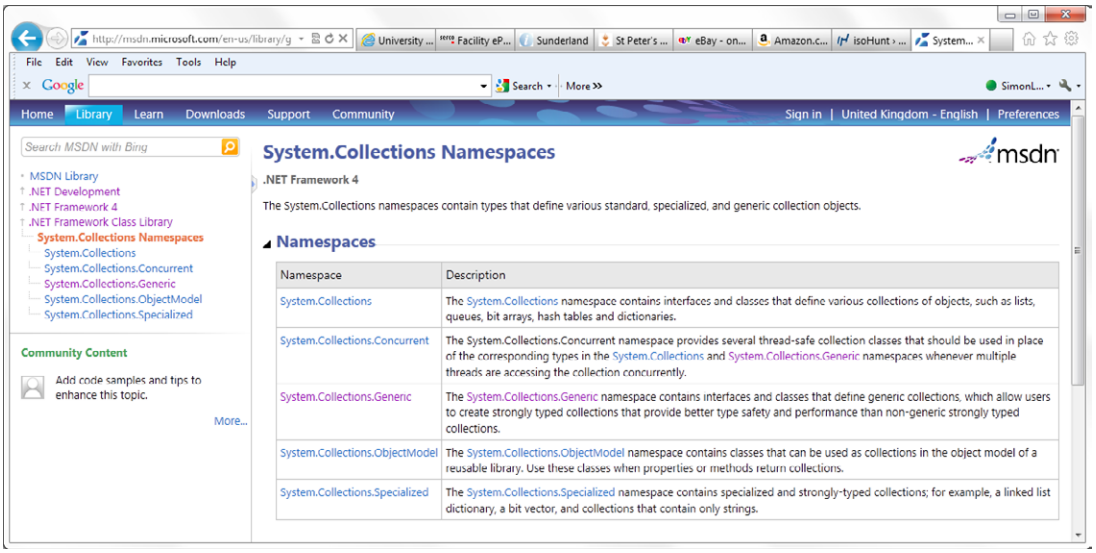
One final complication exist regarding namespace comments.

A room exists and it is possible to go outside of that room and put a sign on the door. A building exists and we can do the same i.e. go outside and put a sign on the building but while the Universe exists we cannot go outside and put a sign on it … because outside does not exist!

A similar problem exists with namespace comments.

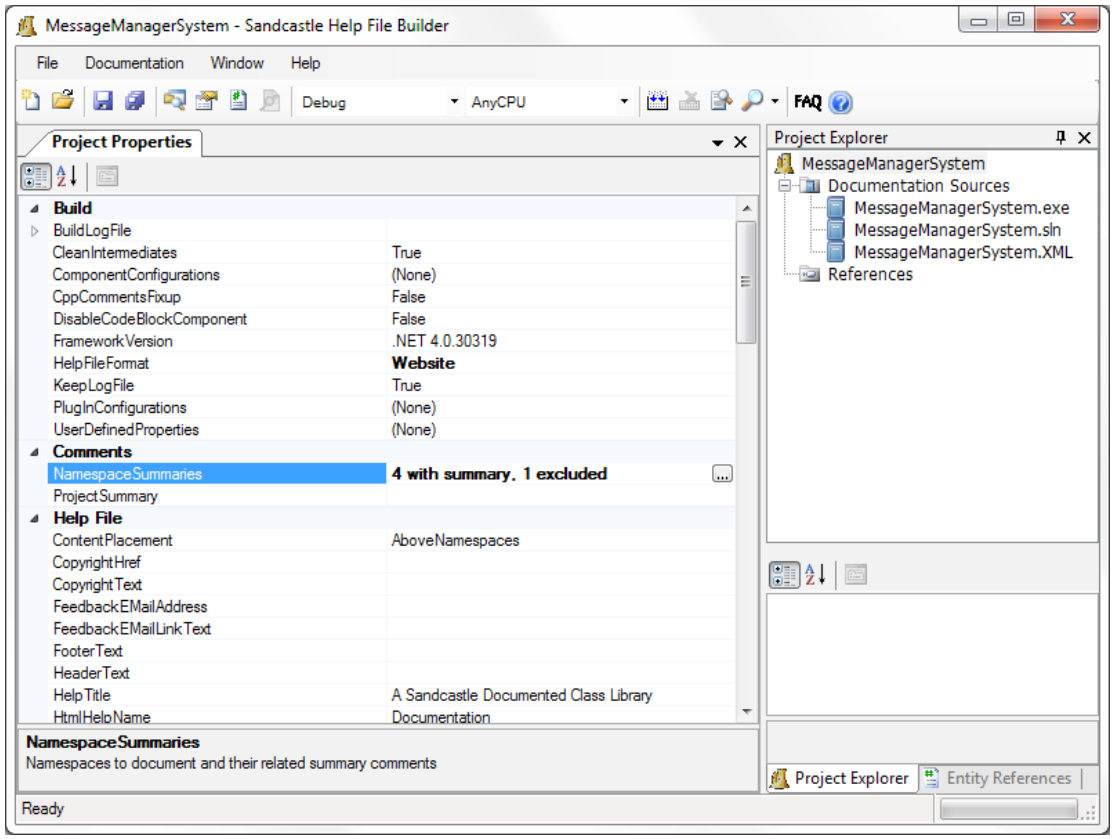
We know where a class starts in our code so we can put an XML comment at the start to describe that class. Similarly we can put comments at the start of methods to describe those methods… but while namespaces exist we cannot define the start of these. So if we segment our larger systems into namespaces, as we should, we cannot put comments at the start of a namespace to describe that namespace.

If we look at part of the MSDN documentation (see figure below) we see that it describes name spaces and we should do the same for our programs.



But if we cannot define the start of a namespace we must put those comments elsewhere.

SHFB provides a solution to this problem by allowing us to enter the namespace comments for our system directly into SHFB.



While it may involve several pieces of software and take a while to set up the ability to generate automatic documentation can save hours and hours of effort. What is more each time we edit and amend our programs we can then generate automatic documentation within a few button pushes.

Other programmers may need to amend and update our the programs we create. As professional programmers we have a duty to the to provide documentation to make this task as easy as possible. Automatic tools such as SHFB and GhostDoc can help us do this.

8.8 Summary

We can go 'back to basics' creating C# programs using a text editor and standalone compilers (freely available) but the use of a professional IDE can offer additional facilities to support programmers.

Specialist tools are available for aspects of development such as GUI design, diagramming and documentation but some IDEs go beyond the basics and provide some of these facilities in built into the IDE.

Visual Studio Express and SharpDevelop are two free powerful IDEs which support the development of C# and other .NET programs.

These tools can initially seem daunting as they provides extensive support for professional development including code formatting and refactoring though online help does exist and both tools provide some level of automatic code generate, e.g. the main method, to make the job of the programmer easier.

Professional programmers have a duty to create up to date documentation. The SHFB software is an extremely useful and timesaving by helping to create this documentation but it does require the programmer to inserting meaningful XML comments into their code (for all classes and all public methods).

GhostDoc be help the programmer by adding XML comments though they will need some manual editing.

9 Creating And Using Exceptions

Introduction

If the reader has written C# programs that make use of the file handling facilities they will have probably written code to catch exceptions i.e. they will have used try\catch blocks. This chapter explains the importance of creating your own exceptions and shows how to do this by extending the Exception Class and using the 'Throw' mechanism.

Objectives

By the end of this chapter you will be able to….

- Appreciate the importance of exceptions
- Understand how to create your own exceptions and
- Understand how to throw these exceptions.

This chapter consists of six sections :-

1) Understanding the Importance of Exceptions
2) Kinds of Exception
3) Extending the ApplicationException Class
4) Throwing Exceptions
5) Catching Exceptions
6) Summary

9.1 Understanding the Importance of Exceptions

Exception handling is a critical part of writing C# programs. The authors of the file handling classes within the C# language knew this and created routines that made use of C# exception handling facilities – but are these really important? and do these facilities matter to programmers who write their own applications using C#?

Activity 1

Imagine part of a banking program made up of three classes, and three methods as shown below…..

The system shown above is driven by the BankManager class. The AwardLoan() method is invoked, either via the interface or from another method. This method is intended to accept or reject a loan application.

The BookofClients class maintains a set of account holders…people are added to this set if they open an account and of course they can be removed. However the only method of interest to us is the GetClient() method. This method requires a string parameter (a client ID) and either returns a client object (if the client has an account at that bank) – or returns NULL (if the client does not exist).

The Client class has only one method of interest DetermineCreditRating(). This method is invoked to determine a clients credit rating – this is used by the BankManager class to decide if a loan should be approved or not.

Considering the scenario above look at the snippet of code below …

```
Client c = listOfClients.GetClient(clientID) ;
c.DetermineCreditRating();
```

This fragment of code would exist in the AwardLoan() method. Firstly it would invoke the GetClient() method, passing a client ID as a parameter. This method would return the appropriate client object (assuming of course that a client with this ID exists) which is then stored in a local variable 'c'. Having obtained a client the DetermineCreditRating() method would be invoked on this client.

Look at these two lines of code. Can you identify any potential problems with them?

Feedback 1

If a client with the specified ID exists this code above will work. However if a client does not exist with the specified ID the GetClient() method will return NULL.

The second line of code would then cause a run time error (specifically a NullReferenceException) as it tries to invoke the DetermineCreditRating() method on a non existent client and the program would crash at this point.

Activity 2

Consider the following amendment to this code and decide if this would fix the problem.

```
Client c = listOfClients.GetClient(pClientID) ;
If (c !=NULL) {
     c.DetermineCreditRating();
}
```

Feedback 2

If the code was amended to allow for the possible NULL value returned it would work – however this protection is insecure as it relies on the programmer to spot this potential critical error.

When writing the GetClient() method the author was fully aware that a client may not be found and in this case decided to return a NULL value. However this relies on every programmer who ever uses this method to recognise and protect against this eventuality.

If any programmer using this method failed to protect against a NULL return then their program could crash – potentially in this case losing the bank large sums of money. Of course in other applications, such as an aircraft control system, a program crash could have life threatening results.

A more secure programming method is required to ensure that that a potential crash situation is always dealt with!

Such a mechanism exists - it is a mechanism called 'exceptions'.

By using this mechanism we can trap any potential errors in our code – preventing and managing crash situations.

In the situation above we could write code that would catch NullReferenceExceptions and decide how our program should respond to these. Of course we may not know specifically what generated the NullReferenceException so we may not know how our program should respond but we could at least shut down our program in a neat and tidy way, explaining to the user we are doing this because of the error generated. Doing this would be far better than allowing our program to crash without explanation.

In the example above we could be much cleverer still. The GetClient() method could be written in such a way that it generates a new type of exception 'UnkownClient' exception. We could catch this specific exception and knowing exactly what the problem is we could define a much better response…. In this case we could explain that no client exists for the specified ID, we could then ask the user to re-enter their client ID and the program could continue.

9.2 Kinds of Exception

In order to generate meaningful exceptions we need to extend the Exception base class built into the .NET framework.

The Exception class has already been extended to create a SystemException class and an ApplicationException class.

The SystemException class is used to generate exceptions within the .NET framework such as NullReferenceException.

We should use the ApplicationException class when generating exceptions within our application such as UnknownClientException.

Subclasses of ApplicationException are used to catch and deal with potential problems when running our applications. To do this we must 1) create appropriate sub classes of ApplicationException 2) generate exception objects using a **throw** clause when appropriate and 3) catch and deal with these exceptions using a **try/catch** block.

9.3 Extending the ApplicationException Class

When writing our own methods we should look for potential failure situations (e.g. value that cannot be returned, errors that may occur in calculation etc). When a potential error occurs we should generate an 'ApplicationException' object i.e. an object of the ApplicationException class. However it is best to first define a subclass of the ApplicationException i.e. to create a specialised class and throw an object of this subtype.

A new exception is just like any new class in this case it is a subclass of ApplicationException.

In the case above an error could occur if no client is found with a specified ID. Therefore we could create a new exception class called 'UnknownClientException'.

There are several overloaded constructors for the ApplicationException class. One of these requires a String and it uses this to nitialize a new instance of the ApplicationException class with a specified error message.

Exception classes have a 'Message' property we can make use of.

Thus we could create a subclass called UnknownClientException and override this constructor. When we create an object of this class we can use this string to give details of the problem that caused the exception to be generated. This string will be stored and later accessed via the Message property.

The code to create this new class is given below…..

```
public class UnknownClientException :ApplicationException
{
     public UnknownClientException(String message)
          base(message)
     {
     }
}
```

In some respects this looks rather odd. Here we are creating a subclass of ApplicationException but our subclass does not contain any new methods – nor does it override any existing methods. Thus its functionality is identical to the superclass – however it is a subtype with a meaningful and descriptive name.

If sublasses of ApplicationException did not exist we would only be able to catch the most general type of exception i.e an ApplicationException object. Thus we would only be able to write a catch block that would catch every single type of exception generated by our system.

Having defined a subclass we instead have a choice… a) we could define a catch block to catch objects of the most general type 'Exception' i.e. it would catch ALL exceptions or b) we could define a catch block that would catch ALL application generated exceptions or c) we can be more specific and catch only UnknownClientExceptions and ignore other types of exception.

By looking online at the MSDN library we can see that many predefined subclasses of SystemException already exist. There are many of these including :-

- ArithmeticException
    - DiveideByZeroException
    - OverflowException
- IOException
    - DriveNotFoundException
    - FileLoadException
    - FileNotFoundException
    - PathTooLongException
    -
- InvalidCastException
- InvalidDataException
- InvalidOperationException

Thus we could

a) write a catch block that would react to ALL of these exceptions by catching a SystemException, or we could
b) write a catch block that would catch any type of input \ output exceptions, IOExceptions, and ignore all others or
c) we could be even more specific and catch only FileNotFoundExceptions.

Catching specific exceptions allows us to take specific remedial action e.g. given a FileNotFoundException we could explain to the user of a program that the file was missing and allow them to specify an alternative file.

Catching general exceptions allows us to ensure no potentially fatal error causes our program to crash though we may not be able to take very useful remedial action e.g. given any Exception we could close our program down but without knowing what caused the exception we could not take specific remedial action.

In exactly the same way we can define sub classes of ApplicationException and catch these exception as and when appropriate.

9.4 Throwing Exceptions

Having defined our own exception classes we must then ensure our methods generate, or throw, these exceptions when appropriate. For example we would instruct a GetClient() method to throw an UnknownClientException when a client cannot be found with the specified ID.

To do this we must create an object of UnknownClientException using the keyword new. When doing so we can pass an error message as a parameter to the constructor of the exception class as shown in the code below….

```
new UnknownClientException("ClientBook.GetClient(): unknown client ID:" + clientID);
```

The code above generates an instance of the UnknownClientException class and provides the constructor with a String message. This message specifies the name of the Class / Method where the exception was generated from and provides some additional information that can inform the user about the circumstances that caused the error e.g. the clients ID.

Having generated an exception object we use the keyword 'throw' to throw this exception at the appropriate point within the body of the method.

```
public Client GetClient(int clientID)
{
     .
     .
     .
     code missing
     .
     .
     .
     throw new UnknownClientException("ClientBook.GetClient(): unknown client ID:" + clientID);

}
```

In the example above if a client is found the method will return the client object (though this code is not shown). However if a client has not been found the constructor for UnknownClientException is invoked, using 'new'. This constructor requires a String parameter – and the string we are passing here is an error message that is trying to be informative and helpful. The message is specifying :-

- the class which generated the exception (i.e. ClientBook),
- the method within this class (i.e. GetClient()),
- some text which explains what caused the exception and
- the value of the parameter for which a client could not be found.

By defining an UnkownClientException we are enabling methods calling this one to catch an deal with potentially serious errors.

9.5 Catching Exceptions

Having specified to the compiler that this method may generate an exception we are enabling other programmers to protect against potentially critical errors by placing calls to this method within a try / catch block. The code in the try block will be terminated if an exception is generated and the code in the catch block will be initiated instead.

Thus in the example above the AwardLoan() method can decide what to do if no client with the specified ID is found…..

```
try
{
      Client c = listOfClients.GetClient(clientID) ;
      c.determineCreditRating();

       // add code to award or reject a loan application based on this
          credit rating

}

catch (UnknownClientException uce)
{
     Console.WriteLine("INTERNAL ERROR IN BankManager.AwardLoan()\n"
                              + "Exception details: " + uce.Message);

}
```

Now, instead of crashing when a client with a specified ID is not found, the UnknownClientException we have deliberately thrown will be handled by the CLR engine which will terminate the code in the try block and invoke the code in the catch block, which in this case will display a message warning the user about the problem.

9.6 Summary

Exceptions provide a mechanism to deal with abnormal situations which occur during program execution.

When writing classes and methods, which may become part of a large application, we should create sub classes of the class ApplicationException and throw exception objects when appropriate.

By making use of the exception mechanism we are protecting against potentially life threatening program failure.

The exception mechanism will allow other programmers who use our methods recognise and deal with error situations.

When exceptions are generated the code in a catch block will be initiated – this code could take remedial action or terminate the program generating an appropriate error message. In either case at least the program doesn't just 'stop'.

An example of use of exceptions in a fully working system can be found in the case study Chapter 11.

10 Agile Programming

While a detailed discussion of Agile development methods is beyond the scope of this book, this chapter will explain the claims made by proponents of agile programming methods and show how modern IDE's (such as Visual Studio) offers tools to support agile programming. In particular we will examine refactoring and the automatic testing framework within Visual Studio.

Objectives

By the end of this chapter you will be able to

- Appreciate the importance of the claims made for Agile development
- Understand the need for refactoring and how a modern IDE supports this
- Understand the advantages of Unit testing and
- Understand how to create automated test cases.
- Understand the claims made for Test driven Development

This chapter consists of fifteen sections :-

1) Agile Approaches
2) Refactoring
3) Examples of Refactoring
4) Support for Refactoring
5) Unit Testing
6) Automated Unit Testing
7) Regression Testing
8) Unit Testing in Visual Studio
9) Examples of Assertions
10) Several Test Examples
11) Running Tests
12) Test Driven Development (TDD)
13) TDD Cycles
14) Claims for TDD
15) Summary

10.1 Agile Approaches

Traditional development approaches emphasized detailed advance planning and a linear progression through the software lifecycle *Code late, get it right first time* (Really??)

Recent 'agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution *Code early, fix and improve it as you go along.*

This is a very hot topic in Software Engineering circles at the moment, and as with all such developments it has its share of zealots and ideologues!

Is the waterfall lifecycle model really successful in enabling large, complex projects to proceed from start to finish without ever looking back? Advocates of agile approaches contend that these better fit the reality of software development.

However agile programming requires tools that will enable software to change and evolve. Two specific tools provided by modern IDEs that support agile programming are refactoring and testing tools.

10.2 Refactoring

A key element of 'agile' approaches is 'refactoring'. This technique accepts that some early design and implementation decisions will turn out to be poor, or at least less than ideal.

Refactoring means changing a system to improve its design and implementation quality without altering its functionality (in traditional development such work was termed 'preventive maintenance').

Although the idea of structurally improving existing software is not new, the difference is as follows. In traditional development it was seen as a remedial action taken when the software design quality had degraded, usually as a result of phases of functional modification and extension. In agile methodologies refactoring is regarded as a natural healthy part of the development process.

10.3 Examples of Refactoring

During the development process a programmer may realise that a variable within a program has been badly named. However changing this is not a trivial task.

Changing a local variable will only require changes in one particular method – if a variable with the same name exists in a different method this will not require changing.

Alternatively changing a public class variable could require changes throughout the system (one reason why the use of public variables are not encouraged).

Thus implementing a seemingly trivial change requires an understanding of the consequences of that change.

Other more complex changes may also be required. These include..

- Renaming an identifier everywhere it occurs
- Moving a method from one class to another
- Splitting out code from one method into a separate method
- Changing the parameter list of a method
- Rearranging the position of class members in an inheritance hierarchy

10.4 Support for Refactoring

Even the simplest refactoring operation, e.g. renaming a class, method, or variable, requires careful analysis to make sure all the necessary changes are made consistently.

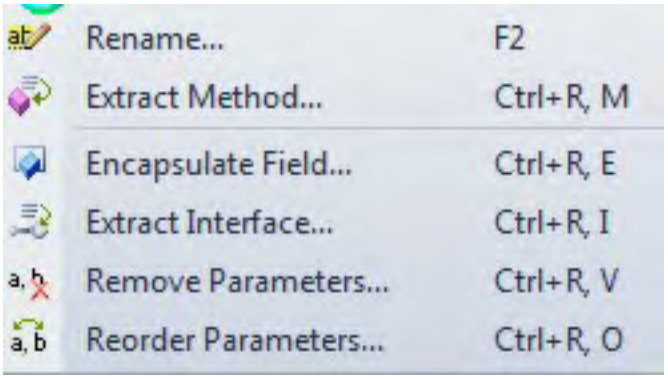
Visual Studio and many other IDEs provide sophisticated automatic support for this activity.

Don't confuse this with a simple text editor find/replace – Visual Studio understands the C# syntax and works intelligently… e.g. if you have local variables with the same name in two different methods and rename one of them, Visual Studio knows that the other is a different variable and does not change it. However if you rename a public instance variable this may require changes in other classes and even in other namespaces as methods from these classes may access and use this variable.

Changing methods to reorder or remove parameters are also refactoring activities. Doing this will require changes to be made wherever these methods are called throughout the system. Visual Studio will automatically change the method calls as appropriate.

Visual Studio provides automated support for the creation of an Interface. To do this it extracts an Interface from a class i.e. given a class it can define an interface based on specified features of that class it will then automatically change that class to define the fact that it implements that new interface. This is a significant restructuring activity that allows us then to create new classes which implement the same interface!

The screen shot below shows the refactoring options provided by the Visual Studio IDE.



Another essential facility provided by modern IDEs is automated testing tools.

10.5 Unit Testing

Testing the individual methods of a class in isolation from their eventual context within the system under construction is known as Unit Testing.

This testing is generally 'wrapped into' the implementation process:

- Write class
- Write tests
- Run tests, debugging as necessary

A unit test should be independent i.e. it should not require other methods or classes within the system to work.

10.6 Automated Unit Testing

To save time we want to automate unit tests but this will not work if the tests require a human to type in test data or if we need a human to check the programs output. Hence to enable automatic testing we need to set up the test data and we need an automated method for checking the program outputs.

Test cases follow a similar anatomy, no matter which testing framework is used, referred to as the arrange-act-assert cycle. Firstly we 'arrange' the test by setting up appropriate test data. Then we 'act' i.e. we run the test. Finally we 'assert' what the expected output should be. The computer can then compare the actual output against the expected output. If there is a difference then the test indicates that the code is faulty.

Tools and frameworks are available to automate the unit testing process. Using such a tool generally requires a little more effort than running tests once manually and significant benefits arise from the ability to re-run the tests as often as desired just by pushing a button.

This is a great aid to the 'regression testing' which must be undertaken whenever previously tested code is modified.

Regression testing was developed long before agile methods were proposed and automated unit testing supports this. However automated unit testing also supports Test Driven Development processes and these play a major role in agile software development methods.

We will explore the Test Driven Development processes within this chapter but before doing so we will first explore conventional regression testing and the support offered for this via the unit testing framework within Visual Studio Professional edition. While this support is not available in the Express edition other tools do support unit testing… though the mechanics will be different from those described here the principles will be the same.

10.7 Regression Testing

All large software systems need to be adapted to meet changing business needs. Many large systems may have been in use for a decade or more. Over the years the software will need to be updated and improved many times to meet the ever changing needs of the client. For this reason regression testing is required. By running regression tests we want to ensure that changes to the code to not 'break' existing functionality. To do this as we write classes we must also write test cases that demonstrate these classes work. Thus we follow a process of …

- Write class
- Write tests
- Run tests, debugging as necessary
- Write more classes
- …

As we decide its necessary to change some of the earlier classes (or classes which they depend on) due to bugs, changing user requirements, or refactoring, we need to re-run all previous tests to check the new code still passes the older tests. Without regression testing any modification of existing code is extremely hazardous!

As we regularly need to re-run sets of test cases it is helpful, and hugely timesaving, to have automated testing facilities such as those that exist within the professional edition of Visual Studio.

10.8 Unit Testing in Visual Studio

Visual Studio includes a widely used unit testing framework. This framework requires us to set up the test cases – however once these have been set up a thousand test cases can be run at the push of a button – and the same set of test cases can be re-run every time the program is amended.

The system will run the tests and highlight which tests pass and which fail.

Note that tests that which have 'failed' actually indicate a failure in the program being tested…. You could argue that in showing this failure the test has in fact been successful.

A summary of the process if explained here ..

1) Firstly set up a project to store all of the tests for a system
2) Within this create a test fixture for each of the classes we are testing
3) Within each test fixture create multiple tests. Several are probably required for each method being tested.
4) Setup the tests i.e. define any initialisation that must be done before the tests are run
5) Teardown the tests i.e. define anything that should be done after the tests have been performed.
6) Run the tests

A full treatment of this framework and how to use it for unit testing can be found at:-
http://msdn.microsoft.com/en-us/library/ms182515%28v=VS.90%29.aspx

There is a naming convention that makes it easy to relate the tests to the code being tested :-

1) The project used to store the tests is named after the system being tested (e.g. for example a project storing tests for a bank system could be called BankSystemTests)
2) The test fixture is named using the name of the class being tested (e.g. assuming within the bank system there is a class called 'Client' a test fixture would be called TestFixture_ClientTests).
3) The tests are named using the method name being tested _ the test being applied _ the expected result (e.g. a method called AddMoney may have a test as follows…AddMoney_TestAdd5Pounds_BalanceEquals10)

In Visual Studio following the menu Test \ New Test \ Basic Test will set up a testing suite capable of testing C# code. Firstly The name of the test fixture will be requested and if a test project has not previously been set up a name for this will then be requested. Individual tests will each need to be named as they are added to the test fixture.

Having created a test fixture we need to set up the tests and write the tests themselves. As part of this we need to specify the correct behaviour of the code being tested. We do this using Assert…() methods which must be true for the test to pass.

We also make use of attributes to provide essential information to Visual Studio so that it can run the tests….

- The [TestClass] attribute indicates that the test class will indeed act as a test fixture. Thus this attribute must be placed at the start of each test fixture.

```
[TestClass]
public class TestFixture_ClientTests
{
.
.
.
```

- The [TestInitialize] attribute declares a method that is run before every test case is executed.
- Test cases are methods that must be preceded by the [TestMethod] attribute.
- The [TestCleanup] attribute declares a method that is run after every test case is executed.
- The [ExpectedException(typeof(…))] attribute allows us to specify that we are expecting the code being tested to generate an exception.

We will see examples of each of these shortly.

10.9 Examples of Assertions

When setting up test cases we make assertions. An assertion is a statement which should be true if the code has functioned correctly. Example of assertion include …

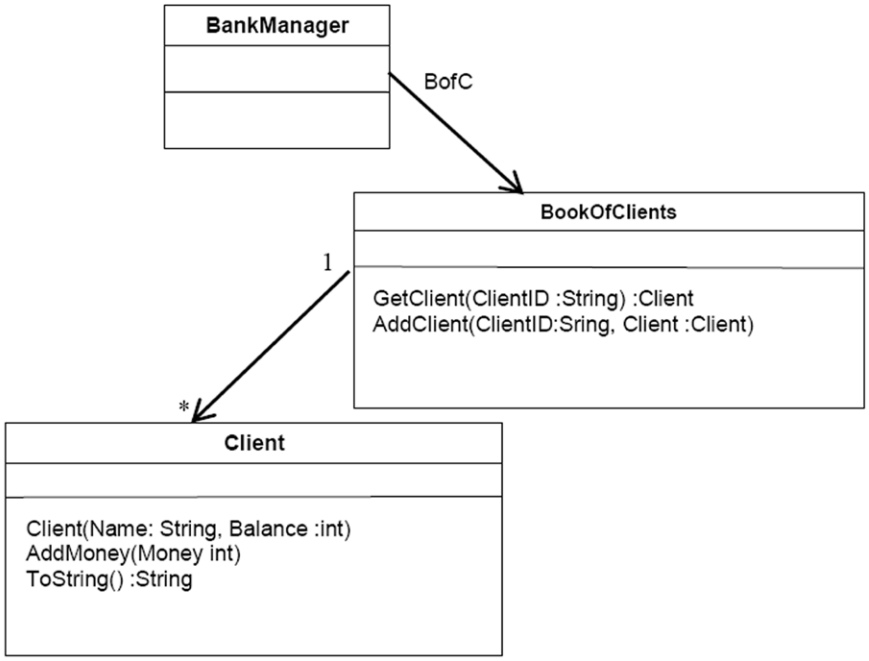
- Assert.AreEqual(…)
- Assert.AreNotEqual(…)
- Assert.Fail()
- Assert.IsTrue()
- Assert.IsFalse(…)

Assert.AreEqual() and Assert.Faill() will be adequate for many testing purposes, as we will see here.

Assert.Faill() indicates that having reached this line means the test has failed! While we say the test has failed this is not strictly true. It is not the test that has failed but our code that has failed. The test successfully found an error in our code.

10.10 Several Test Examples

To illustrate the testing framework we will create several test cases to test the functionality of a BankManagment system as represented below :-



In this system a BankManager class maintains a book of clients (BofC). Client objects can be added by the AddClient() method which requires an ID for that client and the client object to be added. Clients can be retrieved via the GetClient() method which requires a ClientID as a parameter and returns a client object (if one exists) or generates an exception (if a client with the specified ID does not exist).

The Client class has a constructor that requires the name of the client and the clients balance .

For simplicity sake we are assuming each client of the bank only has one account (obviously this is not a realistic system) and we do not need an account number as each client will have a unique ID.

We will specifically test the ability to…

- Add a client to the BookOfClients
- Trying to lookup a non-existent client
- Increase a Clients balance by invoking the AddMoney() method.

This is of course only a small fraction of the test cases which would be needed to thoroughly demonstrate the correct operation of all classes and all methods within the system.

Testing adding a client

To test a client can be added we need to

1) Set up the test by creating a new empty BookOfClients
2) Create a new client and add this to the BookOfClients
3) Check that the client has been added successfully by trying to retrieve the client just added (this of course should work) and finally
4) We need to test that the client retrieved has the same attributes as the client we just added – to ensure it was not corrupted in the process.

Firstly we initialise the test by creating a new empty BookOfClients object (BofC) and we clean up the test by setting this to null at the end. The code for this is given below:-

```
[TestInitialize]
public void TestInitialize()
{
    BofC = new BankManager.BookOfClients();
}

[TestCleanup]
public void TestCleanup()
{
    BofC = null;
}
```

Next we create our first test method. This method will work by trying to add a client to the empty object BoFC. If the system generates an exception because this client already exists then we know there is a fault in our code… hence under these circumstances we assert that the test has failed. See the test for this below…

```
[TestMethod]
public void AddClient_TestNewClient_ExceptionShouldNotBeGenerated()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room 1234", "x200", 10);
     try
     {
          BofC.AddClient(1, c);
     }
     catch(BankManager.ClientAlreadyExistsException)
     {
          Assert.Fail("ClientAlreadyExists exception should not be
          thrown for new clients");
     }
}
```

If the test above passes then we know our code has not generated an exception however this does not prove that the client has been added successfully. We need to create other tests to show that we can retrieve the client just added and we need a test to show that in adding / retrieving the client object the attributes have not been corrupted. Multiple tests are required before we can have confidence that the method being tested will actually work!

Note the name of the test above indicates the name of the method being tested followed by a short description of the test being performed and a description of the expected output. Test names like this, while long, are important. By reading a long list of test names we can work out which parts of our system have been adequately tested and which parts have been missed.

In addition to the tests described above we should also try to add a client with the same ID twice. Our system should of course not allow this to happen. Hence the test succeeds if the code prevents a client being added twice.

Conversely if the system does not generate an exception after adding the same client for the second time then the test should indicate that the code has failed. A test for this is given below…

```
[TestMethod]
public void AddClient_TestAddExistingClient()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room
     1234", "x200", 10);
     try
     {
          BofC.AddClient(1, c);
          BofC.AddClient(1, c);
          Assert.Fail("ClientAlreadyExists exception should be thrown
          if client added twice");
     }
     catch (BankManager.ClientAlreadyExistsException)
     {
     }
}
```

An alternative way of writing the test above is to indicate that the test expects an exception to be generated by using the [ExpectedException(…)] attribute. If this exception is generated and interrupts the test then the test has passed. See an alternative version of the test code below…

```
[TestMethod]
[ExpectedException(typeof(BankManager.ClientAlreadyExistsException))]
public void AddClient_TestAddExistingClient()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room
     1234", "x200", 10);
     BofC.AddClient(1, c);
     BofC.AddClient(1, c);
     Assert.Fail("ClientAlreadyExists exception should be thrown if
     client added twice");
}
```

Testing the GetClient method

We must of course test all of the methods in our system including the GetClient() method.

One test case we need to perform is to test that an exception is thrown if we try to retrieve a client that does not exist. To test this we create an new empty BookofClients and try to retrieve a client from this – any client!

In this instance we would hope our system generates an unknown client exception. Since we have initialised our test by creating an empty client book we should not be able to retrieve any clients unless we first add them.

Activity 1

Look at the test code below and decide at which point in this code we could assert the test has shown our system has failed (point A, B, C or D).

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException()
{    //point A
     try
     {
          // point B
          BofC.GetClient(1);
          // point C
     }
     catch (BankManager.UnknownClientException)
     {
          // point D
     }
}
```

Feedback 1

We cannot determine if the GetClient() method has failed before we have invoked it … hence we cannot place an Assert.Fail() at point A or B.

When we do invoke this method an exception should be generated as our client book is empty. The try block should be terminated at this point and the catch block initiated hence if the code reaches point D the test has succeeded not failed. However if the code reaches point C it indicates that the expected exception was not generated and hence we can assert that the GetClient() method has failed at this point.

The complete code for this test is given below….

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException ()
{
     try
     {
          BofC.GetClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
     }
     catch (BankManager.UnknownClientException)
     {
     }
}
```

As well as testing our BookOfClients class we should of course test the other classes in our system.

Testing the Client Class

As well as testing the BookOfClients class we should test our Client class. The tests for this class should be in a different test fixture and will need to initialise these tests as well….

```
[TestClass]
public class TestFixture_ClientTests
{

     private MessageManagerSystem.Clients.Client c;

     [TestInitialize]
     public void TestInitialize()
     {
          c = new BankManager.Client("Simon", 10);
     }

     [TestCleanup]
     public void TestCleanup()
     {
          c = null;
     }
```

In the initialisation we have created a new client with an opening balance of 10.

Strictly speaking we should have written the Client tests before the BookOfClient tests as we need client objects to test the BookOfClients. In VisualStudio it is possible to ensure these tests are run first.

Below is a simple test to test the AddMoney() method…

```
[TestMethod]
public void AddMoney_TestAdd10ToTheBalance_FinalBalShouldBe20()
{
     c.AddMoney(10);
     Assert.AreEqual(20, c.Money, "Balance after adding 10 is not as
     expected. Expected: 20 Actual: "+c.Money);
}
```

The test above uses the Assert.AreEqual() method. If the first two parameters are equal then our code has passed the test. If they are not equal then our code has failed and the third parameter is the error message to be displayed. To be as helpful as possible the error message specified the balance we expected and the actual balance after the AddMoney() method was invoked.

In all of the tests we have written if the test method ends without failing any assertions then the test is passed.

Testing the ToString() method

One way of implicitly testing that ALL the attributes a client have been stored correctly is to test the ToString() method returns the value expected. This is a little tricky because the format of the string must match exactly including every space, punctuation symbol, and newline.

The alternative however is to test the value of every property.

[Activity 2

Assuming the ToString() method of the Client class is defined as below create a test method to test the value returned by the ToString() method is as expected.

```
public String ToString() {
        return ("Client name: " + Name + "\nBalance: " + Balance);
}
```

Hint: We have already initialised client tests by creating a client with a name "Simon" and a balance of 10.

Feedback 2

One solution to this exercise is given below.

```
[TestMethod]
public void ToString_TestClientValues()
{
     Assert.AreEqual ("Client name: Simon\nBalance: 10", c.ToString(),
     "String returned is not as expected. Expected: Client name:
     Simon\nBalance: 10 Actual: " + c.ToString());
}
```

Here we assert that the string returned from c.ToString() is equal to the string we are expecting. This is quite tricky because the format of the string must match **exactly** including every space, punctuation symbol, and newline.

This test is of course implicitly testing that ALL the attributes have been stored correctly which would be particularly useful if we used it to check a client has been retrieved from the book of clients correctly.

10.11 Running Tests

Having designed a batch of test cases these can be run as often as required at the push of a button.

To do this in Visual Studio we run all of the tests via the Test menu.

10.12 Test Driven Development (TDD)

While very useful for regression testing automated unit testing also supports Test Driven Development (TDD) which is a technique mainly associated with 'agile' development processes. This has become a hot topic in software engineering

The Test Driven Development approach is to

1) Write the tests (before writing the methods being tested).
2) Set up automated unit testing, which fails because the classes haven't yet been written!
3) Write the classes and methods so the tests pass

This reversal seems strange at first, but many eminent contributors to software engineering debates believe it is a powerful 'paradigm shift'

The task of teaching can be used as an analogy. Which of the following is more focused and leads to better teaching?

- Teach someone everything they should know about a subject and then decide how to test their knowledge or
- Decide specifically what a student should be capable of doing after you have taught them, then decide how to test the student to ensure they are capable of performing this task and finally decide just what you must teach so that they can perform this task and hence pass the test.

It can be argued that the second approach leads to more focussed teaching. In the same way it is argued by some that test driven development leads to simpler code that focuses just on achieving the functionality required.

10.13 TDD Cycles

When undertaking test driven development the test will initially cause a compilation error as the method being tested doesn't exist!

Creating a stub method enables the test to compile but the test will 'fail' because the actual functionality being tested has not been implemented in the method.

We then implement the correct functionality of the method so that the test succeeds.

For a complex method we might have several cycles of: write test, fail, implement functionality, pass, extend test, fail, extend functionality, pass… to build up the solution.

10.14 Claims for TDD

Among the advantages claimed for TDD are:

- testing becomes an intrinsic part of development rather than an often hurried afterthought.
- it encourages programmers to write simple code directly addressing the requirements
- a comprehensive suite of unit tests is compiled in parallel with the code development
- a rapid cycle of "write test, write code, run test", each for a small developmental increment, increases programmer confidence and productivity.

In conventional software lifecycles if a software project is running late financial pressures often result in the software being rushed to market not having been fully tested and debugged. With Test Driven Development this is not possible as the tests are written before the system has been implemented.

10.15 Summary

'Agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution.

Refactoring tools help agile development methods.

Unit testing is an important part of software engineering practice whatever style of development process is adopted.

An automated unit testing framework allows unit tests to be regularly repeated as system development progresses.

Test Driven Development reverses the normal sequence of code and test creation, and plays a major part in 'agile' approaches.

11 Case Study

This chapter will bring together all of the previous chapters showing how these essential concepts work in practise through one example case study.

Objectives

By the end of this chapter you will see

- how a problem description is turned into a UML model (as described in Chapter 6)
- several example of UML diagrams (as described in Chapter 2)
- an example of the use of inheritance and method overriding (as described in Chapter 3)
- an example of polymorphism and see how this enables programs to be extended simply (as described in Chapter 4)
- an example of how generic collections can be effectively used, in particular you will see an example of the use of a set and a dictionary (as described in Chapter 7)
- an example of these collections can be stored in files very simply using the process of serialization (as described in Chapter 7)
- examples of exceptions and exception handling (chapter 9)
- examples of automated unit testing (chapter 10).
- finally you will see the use of the automatic documentation tool (as described in Chapter 8).

The complete working application, developed as described throughout this chapter, is available to download for free as compressed file. This file can be unzipped and the project loaded into Visual Studio 2010.

The express edition of Visual Studio 2010 is available for free download from www.microsoft.com/express/Downloads though this does not include support for the unit testing.

The automatic documentation created to describe the system is available as a set of web pages and can therefore be viewed by any web browser.

This chapter consists of sixteen sections :-

1) The Problem
2) Preliminary Analysis
3) Further Analysis
4) Documenting the design using UML
5) Prototyping the Interface
6) Revising the Design to Accommodate Changing Requirements
7) Packaging the Classes
8) Programming the Message Classes
9) Programming the Client Classes
10) Creating and Handling UnknownClientException
11) Programming the Main classes
12) Programming the Interface
13) Using Test Driven Development and Extending the System
14) Generating the Documentation
15) The Finished System
16) Running the System
17) Conclusions

11.1 The Problem

User requirements analysis is a topic of significant importance to the software engineering community and totally outside the scope of this text. The purpose of this chapter is not to show how requirements are obtained but to show how a problem statement is modelled using OO principles and turned into a complete working system once requirements are gathered.

The problem for which we will design a solution is 'To develop a message management system for a scrolling display board belonging to a small seaside retailer.'

For the purpose of this exercise we will assume preliminary requirements analysis has been performed by interviewing the shop owner, and the workers who would use the system, and from this the following textual description has been generated:-

Rory's Readables is a small shop on the seafront selling a range of convenience goods, especially books and magazines aimed at both the local and tourist trades. It has recently also become a ticket agency for various local entertainment and transport providers.

Rory plans to acquire an LCD message display board mounted above the shopfront which can show scrolling text messages. Rory intends to use this to advertise his own products and offers and also to provide a message display service for fee-paying clients (e.g. private sales, lost and found, staff required etc.)

Each client is given a unique ID number and has a name, an address, a phone number and an amount of credit in 'credit units'. A book of clients is maintained to which clients can be added and in which we can look up a client by their ID.

Each message is for a specific client and comprises the text to be displayed and the number of days for which it should be displayed. The cost of each message is 1 unit per day. No duplicate messages (i.e. the same text for the same client) are permitted.

A set of current messages is to be maintained: new messages can be added, the message set can be displayed on the display board, and at the end of each day a purge is performed – each message has its days remaining decremented and its client's credit reduced by the cost of the message, and any messages which have expired or whose client has no more credit are deleted from the message set.

The software is to be written before the display board is installed – therefore the connection to the board should be via a well-defined interface and a dummy display board implemented in software for testing purposes.

Given the description above this chapter describes how this problem may be analysed, modelled and a solution programmed – thus demonstrating the techniques discussed throughout this book.

11.2 Preliminary Analysis

To perform a preliminary analysis of these requirement as (described in Chapter 6) we must…

- List the nouns and verbs
- Identify things outside the scope of the system
- Identify the synonyms
- Identify the potential c lasses, attributes and methods
- Identify any common characteristics

From reading this description we can see that the first paragraph is purely contextual and does not describe anything specifically related to the software required. This has therefore been ignored.

List of Nouns

From the remaining paragraphs we can list the following nouns or noun phrases:-

- LCD message display board
- shopfront
- scrolling text message
- client
- ID number
- name
- address
- phone number
- credit
- credit unit
- message
- day
- book of clients
- ID
- text
- number of days
- cost of message (units)
- set of current messages
- message set
- display board
- days remaining
- client's credit
- cost of message
- software
- connection
- interface
- dummy display board

List of Verbs

and the following verbs :-

- acquire
- mount
- show
- advertise
- give (a unique ID)
- display
- add (a client)
- look up
- permit (duplicates – NOT!)
- purge
- decrement
- reduce credit
- expire
- delete
- write (the software)
- install
- implement
- test

Outside Scope of System

By identifying things outside the scope of the system we simplify the problem…

- Nouns:
- shopfront
- software
- Verbs:
- acquire, mount (the display board)
- advertise
- give (a unique ID)
- write, install, implement, test (the software)

The shopfront is not part of the system and it is not a part of the system to acquire and mount the displayboard, the ID is assigned by the shop owner – not the system, and writing / installing the software is the job of the programmer – it is not part of the system itself.

Synonyms

The following are synonyms :-

- Nouns:
- LCD message display board = display board
- scrolling text message = message
- ID number = ID
- credit units = client's credit = credit
- set of current messages = message set
- days = number of days = days remaining

- Verbs:
- show = display

By identifying synonyms we avoid needless duplication and confusion in the system.

Potential Classes, Attributes and Methods

Nouns that describe significant entities for which we can identify properties i.e. data and behaviour i.e. methods could become classes within the system. These include :-

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

Nouns that are would be better as attributes of a class rather than becoming a class themselves :-

- For a 'client':
- ID
- name
- address
- phone number
- credit
- For a 'message':
- text
- days remaining
- cost of message

Each of these *could* be modelled as a class (which Client or Message would have as an object attribute), but we decide that each of them is a sufficiently simple piece of information that there is no reason to do so – each one can be a simple attribute (instance variable) of a primitive type (e.g. int) or library class (e.g. String).

This is a **design judgement** – introducing classes for significant entities (Client, Message etc.) which have a set of information and behaviour belonging to them, but not overloading the design with trivially simple classes (e.g. credit which would just contain an 'int' instance variable together with a property or accessor method!).

Verbs describe candidate methods and we should be able to identify the classes these could belong to. For instance :-

- For a 'client':
- decrease credit
- For a 'message':
- decrement days

The other verbs describing potential methods should also be listed:-

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

For each of these the associated class should be identified.

Common Characteristics

The final step in our preliminary analysis is to identify the common characteristics of classes and to identify if these classes should these be linked in an inheritance hierarchy or linked by an interface.

Looking at the list of candidate classes provided we can see that two classes that share common characteristics:-

- DisplayBoard
- DummyDisplayBoard

This either implies these classes should be linked within the system within an inheritance hierarchy or via 'an interface'(see section 4.5

Interfaces).In this case the clue is within the description "These will have a 'connection' to the rest of the system via a 'well-defined interface".

Ultimately our system should display messages in a real display board however it should first be tested on a dummy display board. For this to work the dummy board must implement the same methods as a real display board.

Thus we should define a C# 'interface'. No common code would exist between the two classes – hence why we are not putting these within an inheritance hierarchy. However the dummy board and the real display board should both implement the methods defined via a common interface. When our system is working we could replace the dummy board with the real board which implements the same methods. As the connection with the dummy board is via the interface changing the dummy board with the real display board should have no impact on our system.

From our preliminary analysis of the description we have identified candidate classes, interfaces, methods and attributes. The methods and attributes can be associated with classes.

The classes are : -

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

The Interface is :-

- DisplayBoardControl (a name we have made up)

And the methods include :-

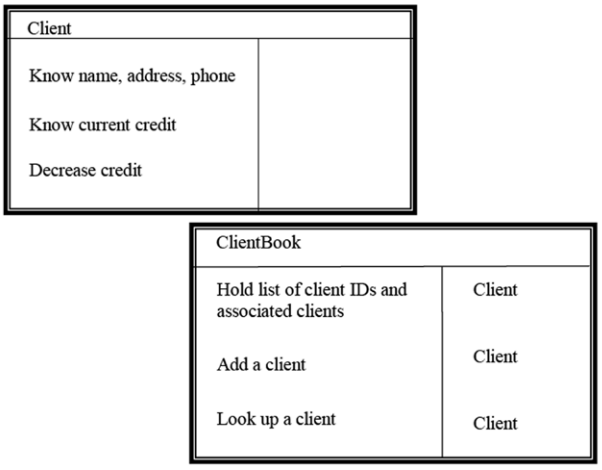
- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

11.3 Further Analysis

We could now document this proposed design using UML diagrams and program a system accordingly. However before doing so it would be better to find any potential faults in our designs as fixing these faults now would be quicker than fixing the faults after time has been spent programming the system. Thus we should now refine our design using CRC cards and elaborate our classes.

CRC cards (see Chapter 6 section 6.10 and 6.11) allow us to role play various scenarios to check if our designs look feasible before refining these designs and documenting them using UML class diagrams.

The two CRC cards below have been developed to describe the Client and ClientBook classes. The panel on the left shows the class responsibilities and the panel on the right shows the classes they are expected to collaborate with.



We can now use these to roleplay, or test out, a scenario. In this case what happens when we get a new client in the shop? Can this client be created and added to the client book?

To do this the system must perform the following actions :-

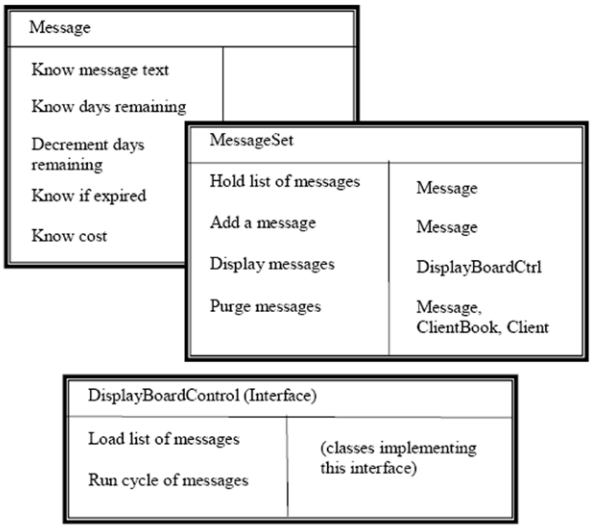
- create a new Client object
- pass it (along with the unique ID to associate with it) to the ClientBook object
- add the client to the client book.

By looking at the CRC cards we can see that :-

- the constructor for Client will be able to create a new client object
- The ClientBook has the capability to add a client and
- the ClientBook can hold the IDs associated with each client.

It would therefore appear that this part of the system will work at least in this respect – of course we need to create CRC cards to describe every class and to test the system with a range of scenarios.

Below are three CRC cards to describe the Message and MessageSet classes and the DisplayBoardControl interface.



What we want to 'test' here is that messages can be created, added to the MessageSet and displayed on the display.

A point of requirements definition occurs here. There are two possibilities regarding the interface to the display board:-

a) we load one message at a time, display it, then load the next message, and so on.
b) we load a collection of messages in one go, then tell the board to display them in sequence which it does autonomously

The correct choice depends on finding out how the real display board actually works.

Note that (a) would mean a simple display board and more complexity for the "Display messages" responsibility of MessageSet, while (b) implies the converse

For this exercise we will assume the answer to this is (b), hence the responsibilities of scrolling through a set of messages will be assigned to the DisplayBoardControl interface.

Looking at these CRC cards it would appear that we can

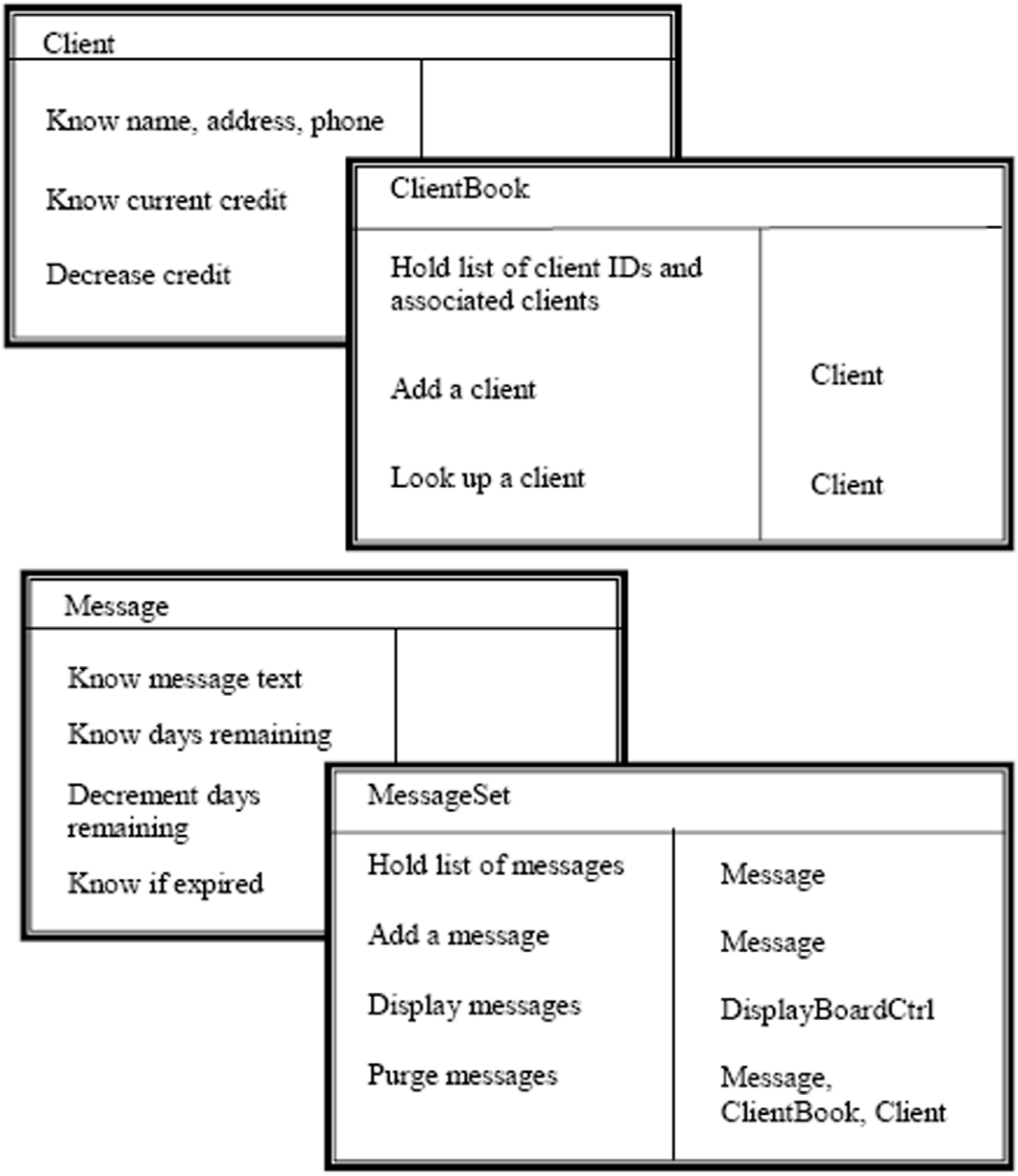
- Create a new message,
- Add this to the message set and
- Display these messages by invoking load 'list of messages' and 'run cycle of messages'

So this part of our design also seems to work.

The Message Purge Scenario

The final scenario that we want to run though here is the message purge scenario. At the end of the day when messages have been displayed the remaining days of the message need to be decremented and the message will need to be deleted if a) the message has run out of days or b) the client has run out of credit.

CRC cards for the classes involved in this have been drawn below…



Activity 1

To purge the messages, the MessageBook cycles through its list of messages reducing the credit for the client who 'owns' this message, decrementing the days remaining for that message and deleting messages when appropriate.

Looking at the CRC cards above work through the following steps and identify any potential problems with these classes :-

For each message
- tell the Message to decrement its days remaining and
- tell the relevant Client to decrease its credit
- ask the Message for its client ID
- ask the Message for its cost
- ask the ClientBook for the client with this ID
- tell the Client to decrease its credit by the cost of the message

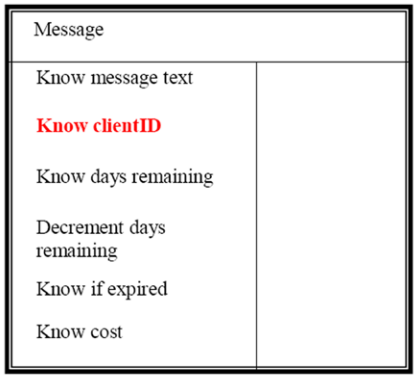
- if either the Client's credit is <= 0 or the Message is now expired
delete the message from the list

Feedback 1

A problem becomes evident when we try to find the client associated with a message as Message does not know the client ID.

We therefore need to add this responsibility to the Message class.

A revised design for the Message class is given below….



By drawing out CRC cards for each class and interface and by role playing a range of scenarios we have checked and revised our plans for the system - we can now refine these and document these using UML diagrams.

11.4 Documenting the design using UML

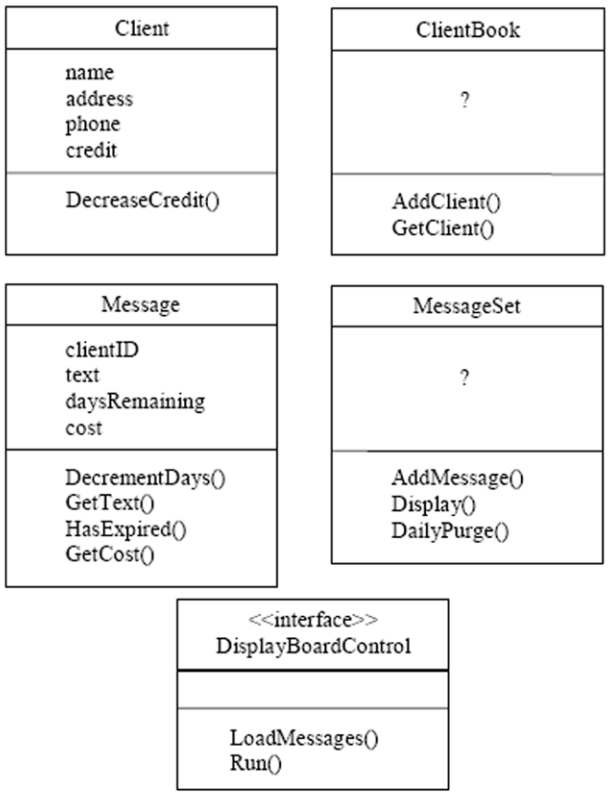
To fully document our designs we need to :-

- Determine in detail what attributes go in each class
- Determine how the classes are related and
- Put classes into appropriate packages.

Elaborating the Classes

Having worked through CRC scenarios we can make an initial assignment of instance variables and methods among our classes, including some accessors and mutators whose necessity has become evident (see diagram below).

Of course as we are going to program this system in C# we will replace our accessor and mutator methods with properties but as this design could be programmed in any OO language we will leave our design showing appropriate accessor and mutator methods.



We don't know of any simple attributes which ClientBook and MessageSet will require, but they will need to be associated with other classes so we still have some work to do there – hence the ?s (which are not an official part of UML).

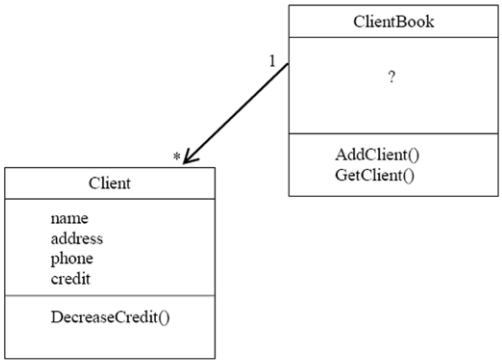
Relationships Between Classes

We can now start to work out how these classes are related.

Starting with ClientBook and Client :- a ClientBook will record details of zero or more clients.

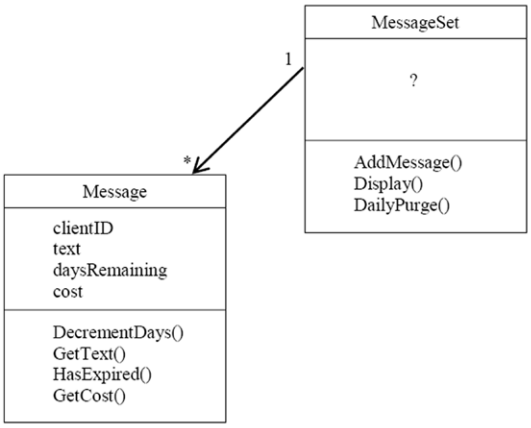
The navigability will be from ClientBook to client because the book "knows about" its Clients (in implementation terms it will have references to them) but the individual Clients will not have references back to the book.

The one-to-many relationship suggests that ClientBook will have a Collection (of some kind) of Clients. The specification states that each Client will have a unique ID thus the collection will in fact be a dictionary where each entry is made up of a pair of values – in this case a clientID (an int) and a Client object. As we are likely to be searching and retrieving clients by their ID far more often than we will be adding and deleting clients the system will be more efficient if we make this dictionary a sorted dictionary, where the clients will be stored in order of their ID. With a sorted dictionary adding and deleting elements will be slower as the process is more complex but retrieving elements will be quicker.



The relationship between MessageSet and Message is very similar to the relationship between ClientBook and Clients.

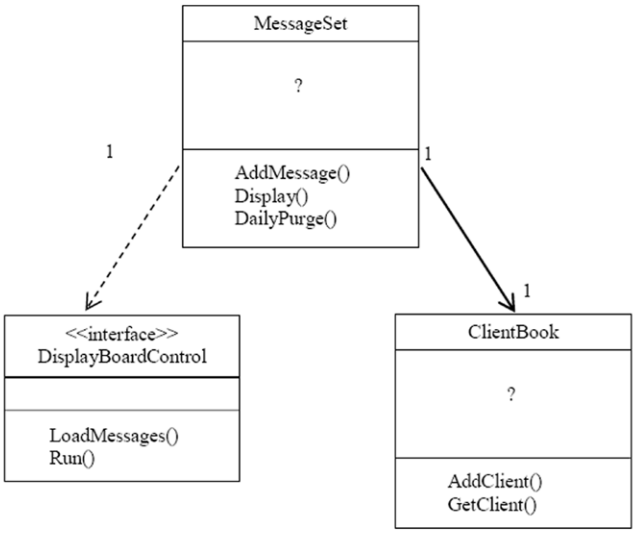
Although MessageSet appears to have no attributes, its one-to-many association with Message again implies an attribute which is a Collection type. The specification states that messages must be unique but does not imply a key value is required thus a simple set will suffice.



Relating the Classes: MessageSet, ClientBook, and DisplayBoardControl

Because MessageSet is responsible for initiating the display of the messages on the display board it has a dependency on a class implementing the DisplayBoardControl interface.

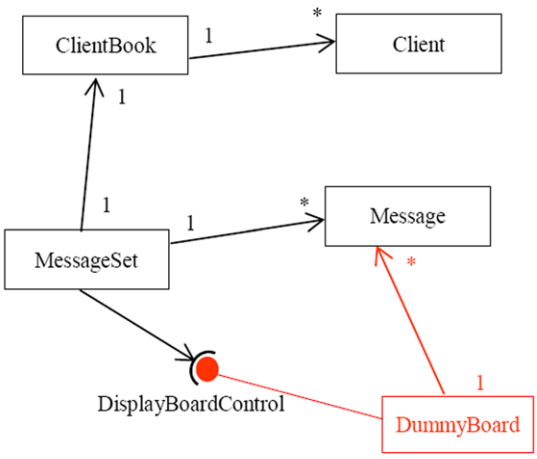
MessageSet also has a relationship with ClientBook because it needs to access and update Client information when the daily purge is carried out. This is shown below..



Relating the Classes Overall

The diagram below shows how all of these classes are related. An additional class, DummyBoard, has been included which will implement the DisplayBoardControl interface for testing purposes.

Since DummyBoard will have a collection of messages loaded it also has a one-to-many relationship with Message.



Note the use of the concise "ball and socket" notation for the DisplayBoardControl interface.

While the classes above will form the heart of the system an additional class will be required to drive and manage the system as a whole.

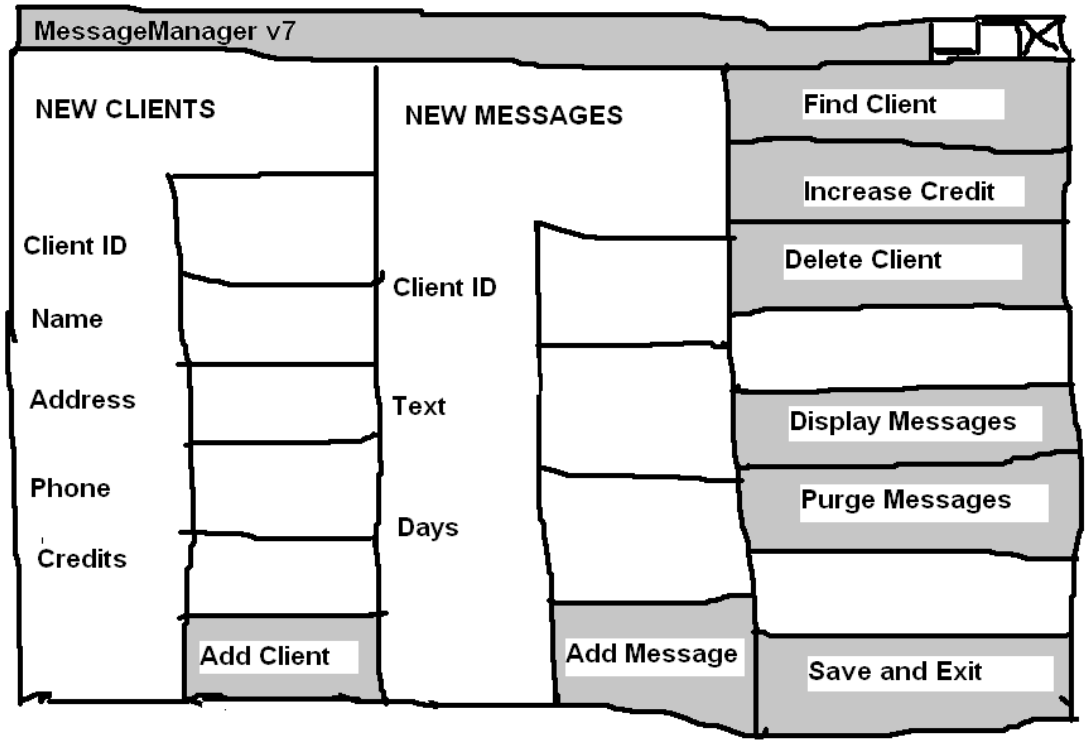
A class 'ApplicationWindow' will be created. This will be the GUI that will run the system and this will allow the user will interact with the system adding clients, messages etc.

Other additional functionality, not specified by the shop owner, is implicitly required. At the end of the day the details of the ClientBook and MessageSet will need to be saved to file. This data will need to be restored next time the system is run as the shop owner will clearly not want to enter details of all the clients every time they run the program. This again will be driven from the interface but the body of the code will be devolved to the relevant classes.

11.5 Prototyping the Interface

While methods for gathering user requirements is beyond the scope of this text – it is always a good idea to prototype an interface and get feedback on this before proceeding with the development.

The figure below shows the proposed interface for this system:-



This is made up of three areas. From left to right these are a) an area for adding new clients, b) and area for adding new messages and c) an area for buttons dedicated to other essential operations.

11.6 Revising the Design to Accommodate Changing Requirements

Changing software requirements are a fact of life and OO programming is intended to help software engineers make program adaptations easier, quicker, cheaper and with less risk of generating errors. The principles of inheritance, method overriding and polymorphism are essential OO features that help in this manner.

In this project when gaining feedback from the shop owner on the prototype interface they comment that they generally like the interface but that they have an additional system requirement :-

> Some messages are 'urgent messages'. These should be highlighted on the display by placing three stars before and after the message and the cost of these messages will be twice the cost of ordinary messages. Other than that urgent messages are just like ordinary messages.

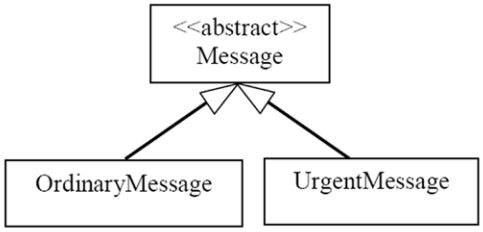
Modifying the interface design to accommodate this change is easy – we can either :-

- create a new panel to accommodate the creation of 'Urgent Messages' or
- since the data required for an urgent message is identical to normal messages we can just add an extra button to the middle panel 'Add Urgent Message'.

But how will these extra requirements impact on the underlying classes within the system?

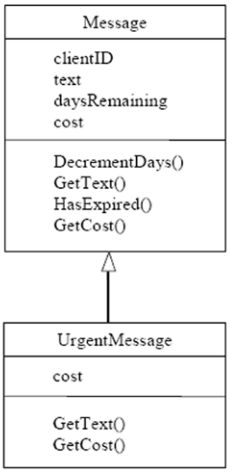
If OO principles work implementing this additional requirement should be relatively simple. Firstly there is clearly a strong relationship between a 'Message' and an 'Urgent Message'

If both classes had some unique features but there was a significant overlap in functionality we could introduce an inheritance hierarchy to deal with this :-

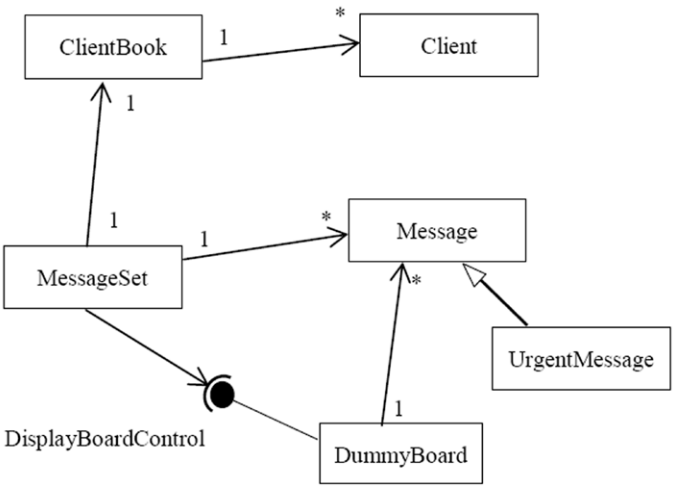


However in this case there are no unique features of an ordinary message – messages have an associated cost, the cost and text can be obtained and new messages can be created. All this is true for urgent messages. An urgent message is just the same as an ordinary message where the text and the cost have been changed slightly. Thus UrgentMessage is a type of Message and can inherit ALL of the features of Message with the cost and text methods being overridden.

Thus the Message and UrgentMessage classes are be related as shown below, with UrgentMessage inheriting all of the values and methods associated with Message but overriding GetCost() and GetText() methods to reflect the different cost and text associated with urgent messages.



A revised class diagram is below. But how will this change impact upon other parts of the system?



Thanks to the operation of polymorphism **this change will have no impact at all on any other part of the system!**

Looking at the class diagram above we can see that MessageSet keeps and manages a set of Messages (DummyBoard also keeps a set of messages - once they have been uploaded for display). But what about UrgentMessages?

Urgent messages are just a specific type of message. When the AddMessage() method is invoked within MessageSet it requires an object of type Message i.e. a message to be added - but an object of the subtype UrgentMessage **is still a 'Message'** so the AddMessage() method would accept an UrgentMessage object.

Therefore, without making any changes at all to MessageSet, MessageSet can maintain a set of all messages to be displayed (both urgent and ordinary)!

Furthermore when the DailyPurge() method is invoked it invokes the GetCost() method on a Message object so that the client can be charged for that message. At run time the Common Language Runtime (CLR) engine will determine whether the object is of type Message or of type UrgentMessage and it will invoke the correct version of the GetCost() method – remember this was overridden in UrgentMessage. This is polymorphism in action!

MessageSet requires messages but, thanks to the application of polymorphism and method overriding, MessageSet will happily deal with any Message subtype as though it were a Message object. If later we decided to create new message types, such as a Christmas or Valentine message, MessageSet would be able to deal with these as well without changing a single line of code!

Thus in this application we are able to extend the system to add the facility for urgent messages by adding only one class and making one small change to the interface.

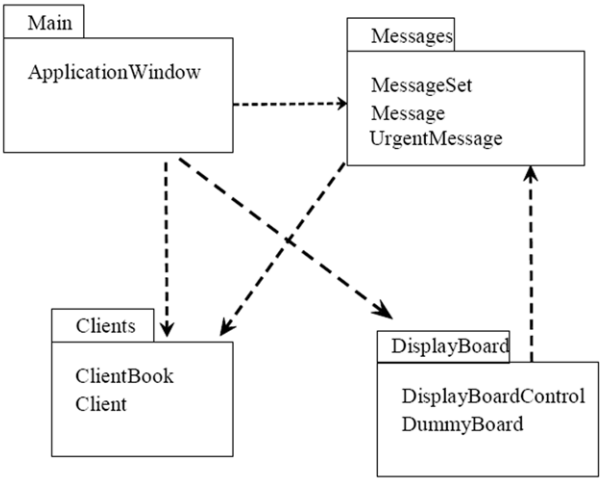
Without the application of polymorphism we would need to have made additional changes to other parts of the system – namely MessageSet and DummyBoard.

Object Orientation has enabled to the system to be extended with minimal effort!

11.7 Packaging the Classes

Large programs should be segmented into packages as this provides an appropriate level of encapsulation and access control (as described in Chapter 2).

The system being used here to demonstrate the theory in this textbook hardly qualifies as large – nonetheless it has been decided to package related classes together as shown below.



This diagram shows the four packages used and the classes within each package. Also shown are associations between the packages. Not surprisingly the main package, which houses the system interface, is associated with all of the other packages – this is because the interface invokes functionality throughout the system.

Having completed the design, and accommodated changing requirements, we can start implementing the system. This will be done in two phases:-

In the first phase a basic system will be implemented which will allow messages and clients to be created, the details written to file and messages to be displayed.

In the second phase the system functionality will be extended to allow clients to be deleted and to allow their credit to be increased. This will be done in a way to allow the demonstration of Test Driven Development (as described in Chapter 10).

11.8 Programming the Message Classes

Message, UrgentMessage and MessageSet are relatively straight forward to program.

Message has various instance variables (String: messageText and int: COST, clientID and daysRemaining). It has appropriate properties to access these private attributes (note only daysRemaining needs a setter) and a constructor to initialize the instance variables. It also has the following methods :-

```
void DecrementDays()   // to reduce the number of days that the message should be displayed for
boolean HasExpired()   // to specify if this message should be removed from the set of messages
String ToString()      // to return the text to be displayed on the displayboard.
```

At some point we will need to store the ClientBook and MessageSet objects to a file. To do this all Client objects and Message objects will also need to be stored hence these classes (including the Message class) will need to be marked as Serializable.

Finally the requirements state that "No duplicate messages (i.e. the same text for the same client) are permitted."

Therefore Message must override the Equals() and GetHashCode() methods to ensure that duplicates will not be permitted when the messages are stored in a Set.

The complete code for this class is given below – though comments have been excluded for the sake of brevity.

The source code for the full system, fully commented, can be viewed by following the instructions near the end of this chapter.

```
namespace MessageManagerSystem.Messages
{
     [Serializable]
     public class Message
     {
          const int COST = 1;
          public virtual int Cost
          {
               get { return COST; }
          }

          private int clientID;
          public int ClientID
          {
               get { return clientID; }
          }

          private String messageText;
          public String MessageText
          {
               get { return messageText; }
          }

          private int daysRemaining;
          public int DaysRemaining
          {
               get { return daysRemaining; }
               set { daysRemaining = value; }
          }

          public Message(int clientID, String text, int daysRemaining)
          {
               this.clientID = clientID;
               this.messageText = text;
               this.daysRemaining = daysRemaining;
          }

          public void DecrementDays()
          {
               daysRemaining--;
          }

          public bool HasExpired()
          {
               return (daysRemaining <= 0);
          }

          public override String ToString()
          {
               return (messageText);
          }
          public override bool Equals(object obj)
          {
               Message m = (Message)obj;
               return (clientID.Equals(m.ClientID) &&
                       messageText.Equals(m.MessageText));
          }

          public override int GetHashCode()
          {
               return (messageText + clientID).GetHashCode();
          }

     }
}
```

The UrgentMessage class is extremely short and sweet as it inherits almost all of its functionality from Message :-

```
namespace MessageManagerSystem.Messages
{

     [Serializable]
     public class UrgentMessage : Message
     {
          const int COST = 2;
          public override int Cost
          {
               get { return COST; }
          }


          public UrgentMessage(int clientID, String text, int
          daysRemaining):base (clientID, text, daysRemaining)
          {
          }

          public override String ToString()
          {
               return ("*** "+ MessageText + " ***");
          }

     }
}
```

Only the 'Cost' property and ToString() methods are overridden as UrgentMessages cost more and the text to be displayed changes. Note to override the Cost property we must mark it as virtual in the Message Class.

The MessageSet class has a one-to-many relationship with Message. This implies a collection type and the fact that duplicate massages are not allowed (at least for the same client) implies the collection should be a Set.

The MessageSet class requires an instance variable to hold the set of messages and it will need access to a ClientBook object as it needs access to the clients when performing a daily purge. The client book object could be stored using an instance variable or passed as a parameter to the DailyPurge() method. As it is only required by this one method the decision was made to pass this as a parameter each time the DailyPurge() is invoked.

A constructor is required to assign a new HashSet() to Messages (the set of messages stored). The following methods are also required :-

void AddMessage(Message msgToAdd)	to add a message to the message set
void display(DisplayBoardControl db)	to display the messages each day on a display board… initially a simulated display board
void DailyPurge(ClientBook clients)	to a) decrement the days remaining at the end of each day for each message, b) charge the client for displaying that message and c) remove all messages that have expired.
private bool ToBeDeleted()	a private method used by the DailyPurge() to denote which messages are to be removed from the message set i.e. those that have expired.

Some of the code from this class is shown below :-

```
namespace MessageManagerSystem.Messages
{
     [Serializable]
     public class MessageSet
     {
          private HashSet<Message> messages;
          public HashSet<Message> Messages
          {
               get { return messages; }
          }

          public MessageSet()
          {
               messages = new HashSet<Message>();
          }

          public void AddMessage(Message msgToAdd)
          {
               messages.Add(msgToAdd);
          }

          public void Display(IDisplayBoardControl db)
          {
               db.LoadMessages(messages);
               db.Run();
          }

          public void DailyPurge(ClientBook clients)
          {
                // code omitted here
          }

          private bool ToBeDeleted(Message m)
          {
               return m.HasExpired();
          }

     }

}
```

The code above shows the creation of a typed collection of 'Message' called Messages and methods to add and display messages.

The method to display messages requires and object of type DisplayBoardControl to be passed as a parameter. Initially a DummyBoard object will be provided however when a real display board is purchased then this object will replace the DummyBoard object. This will have no impact on the code within the Display() method as both objects are of the more general type DisplayBoardControl. This is another example of the application of polymorphism.

The DailyPurge() method was excluded from the code above so we could concentrate on this method now.

The DailyPurge() method performs the following actions:-

For each message

Decrement the days remaining for that message

Find the client who paid for that message

Find the cost of the message and deduct this from that clients credit

For each message

If the message has expired or if the client has run out of credit then
Set the DaysReamining for that message to zero.

Remove all expired messages.

See code below for this…

```
public void DailyPurge(ClientBook clients)
{
    Client client;

    // loop through all current messages and decrement credit and days
    foreach( Message m in messages)
    {

        m.DecrementDays();  // deduct 1 from days remaining for message
        try
        {
            // decrease client credit for this message
            client = clients.GetClient(m.ClientID);
            client.DecreaseCredit(m.Cost);
        }
        catch (UnknownClientException uce)
        {

            MessageBox.Show("INTERNAL ERROR IN MessageSet.Purge()
            \r\nException Details: " + uce.Source + "  \r\nMessage
            details " + uce.Message + ": \r\n");       }

    }
    // loop through all current messages
        and expire those whose client credit <=0
    foreach (Message m in messages)
    {
         try
         {
              client = clients.GetClient(m.ClientID);
              if (client.Credit <= 0)
              {
                   m.DaysRemaining = 0;
              }
         }
         catch (UnknownClientException)
         {
              // Do nothing as unknown clients have been reported to
                 error log already
         }

    }

    // Remove all expired messages
    messages.RemoveWhere(ToBeDeleted);
}
```

Note it is possible that a client could not be found – hence the try catch block in the code above. This will be discussed in the next section.

11.9 Programming the Client Classes

The system needs a Client class, with methods to decrease the client's credit (when a message has been displayed for them). Ultimately it will also need a method to allow a client to pay for and increase their credit but we will not add this functionality in the first version of this system. This class is simple and is very similar to programming the Message class and is therefore not shown here.

Programming the ClientBook class is also similar to programming MessageSet, class however there are a few significant differences :-

- All clients have a ClientID so ClientBook uses a sorted dictionary instead of a Set.
- The method GetClient() could fail if no client exists with the specified ClientID. We need to build in protection in case a client cannot be found.
- The method AddClient() could also fail if a client already exists with the specified ID.

The complete ClientBook class (without comments) is shown below. By downloading the finished program this code can be viewed with embedded comments.

```
namespace MessageManagerSystem.Clients
{
     [Serializable]
     public class ClientBook
     {
          private SortedDictionary<int, Client> clients;
          public  SortedDictionary<int, Client> Clients
          {
               get { return clients; }
          }

          public ClientBook()
          {
               clients = new SortedDictionary<int, Client>();
          }

          public ClientBook(SortedDictionary<int, Client> clients)
          {
               this.clients = clients;
          }

          public void AddClient(int clientID, Client newClient)
          {
               try
               {
                    clients.Add(clientID, newClient);
               }
               catch (ArgumentException)
               {
                    throw new ClientAlreadyExistsException
                    ("ClientBook.AddClient(): a client with this ID
                      already exists in system ID:" + clientID);
               }
          }

          public Client GetClient(int clientID)
          {
               try
               {
                    return clients[clientID];
               }
               catch(KeyNotFoundException)
               {
                    throw new UnknownClientException
                    ("ClientBook.GetClient(): unknown client ID:" +
                     clientID);
               }
          }
     }
}
```

The code above shows the constructors to create a ClientBook object which is a sorted dictionary of ClientID, Client objects and the other methods required by the ClientBook class. Further discussion of this is provided below.

11.10 Creating and Handling UnknownClientException

The GetClient() method will generate a KeyNotFoundException if no client exists with the specified ID. If we do not catch and deal with this exception our program will crash! Furthermore our program will crash as we try to invoke the DecreaseCredit() method without having a client object to invoke this method on.

To protect against this we need to :-

- Create a new kind of exception (as described in Chapter 9) called UnknownClientException
- tell the ClientBook class to throw this exception if a client is not found
- catch and deal with this exception in the DailyPurge() method.

The first step is simple ….

```
public class UnknownClientException :ApplicationException
{
     public UnknownClientException(String message): base(message)
     {
     }
}
```

We next tell the GetClient() method to generate an UnknownClientException if it catches a KeyNotFoundException (as shown below) :-

```
public Client GetClient(int clientID)
{
      try
      {
            return clients[clientID];
       }
       catch(KeyNotFoundException)
       {
            throw new UnknownClientException("ClientBook.GetClient():
                        unknown client ID:" + clientID);
       }
}
```

Under the appropriate condition, we invoke the constructor of the exception using the keyword 'new' and pass a string message required by the constructor. The object returned by the constructor is then 'thrown'.

To be helpful the string specifies the method where this exception was generated from and the clientID for which a client was not found. The DailyPurge() method should catch this exception and hopefully deal with it to prevent a crash situation.

The final step is to catch and deal with UnkownClientException within the DailyPurge() method – as shown in section 11.8 (Programming the Message Classes).

If a client does not exist we could remove the message. However in this case we have chosen to be more cautious since we simply don't know how we have come to have an 'unowned' message.

We have therefore decided that if the message has no recognized client we will not to take any action other than to report the error. The message will continue to be displayed (even without having a client to charge!).

If an unowned message has expired we of course still need to remove it from the display set.

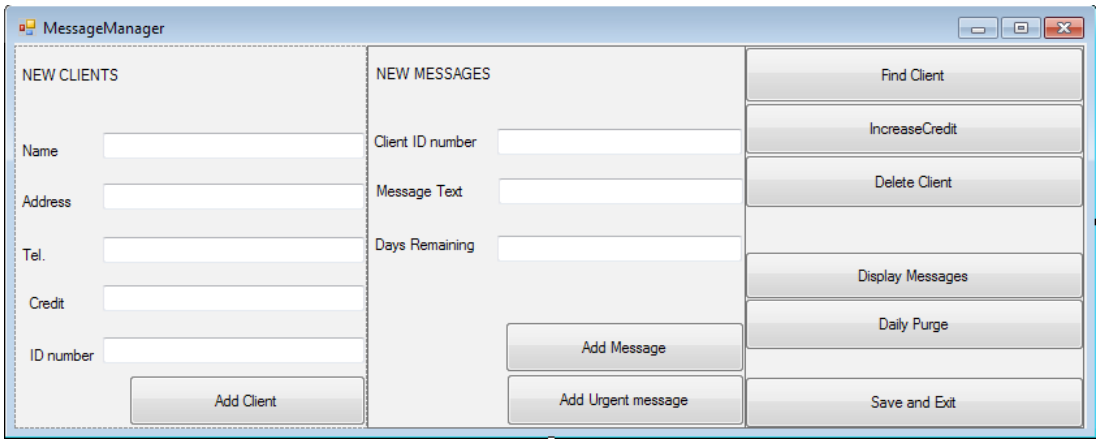
11.11 Programming the Interface

We have still to examine the main application window that will be used to 'drive' the system.

It performs the following functions :-

- it has a permanent reference to the client book and message set.
- it sets up the data file and used for storing the ClientBook and Message Set
- it defines what happens when the system starts and
- it defines what happens when the system shuts down
- it invokes the constructors for the Message, UrgentMessage and Client classes whenever the user wants to add a new message or client to the system.
- it displays the messages on a dummy display board and
- it invokes the DailyPurge() method when requested by the user at the end of each day.

The application window will look as shown below….



Note this includes buttons for increasing a client's credit and deleting a client – functionality we realised some time ago that we needed but functionality that was not added to the first version of this system. We will extend our system to add this additional functionality once a basic system is working.

The ApplicationWindow_Load() method is shown below :-

```
private void ApplicationWindow_Load(object sender, EventArgs e)
{
     clientBook = new ClientBook();
     messageSet = new MessageSet();
     try
     {
         FileStream inFile = new FileStream("ClientAndMessageData",
                                            FileMode.Open, FileAccess.Read);
         BinaryFormatter bFormatter = new BinaryFormatter();
         clientBook = (ClientBook)bFormatter.Deserialize(inFile);
         messageSet = (MessageSet)bFormatter.Deserialize(inFile);
         inFile.Close();
         inFile.Dispose();
     }
     catch (FileNotFoundException)
     {
     }
}
```

The ApplicationWindow_Load() method reconstructs any previously stored ClientBook and MessageSet objects.

The action listener for the SaveAndExit button is shown below….

```
private void btnSaveAndExit_Click(object sender, EventArgs e)
{
     FileStream outFile = new FileStream("ClientAndMessageData",
                                         FileMode.Create, FileAccess.Write);
     BinaryFormatter bFormatter = new BinaryFormatter();


     bFormatter.Serialize(outFile, clientBook);
     bFormatter.Serialize(outFile, messageSet);

     outFile.Close();
     outFile.Dispose();
     this.Close();

}
```

Note how with just four lines of code above we can create an appropriate output stream and save all client book and message set data – this includes details of all clients and all messages. While we had to mark the relevant classes, ClientBook, Client, MessageSet and Message, as serializable we did not have to write any code to save this data to file.

Shown below is the action listener associated with the FindClient button:-

```
private void btnFindClient_Click(object sender, EventArgs e)
{
    Client client;
    try
    {
        InputBox inputBox = new InputBox("Find Client", "Please enter
                                                              clients ID.");
        DialogResult dialogResult = inputBox.ShowDialog();

        if (dialogResult == DialogResult.OK)
        {
               int clientID;
               if (!Int32.TryParse(inputBox.Answer, out clientID))
               {
                    MessageBox.Show("Invalid client ID, please enter
                    integer number.");
                    return;
               }

                client = clientBook.GetClient(clientID);
                MessageBox.Show(client.ToString());
          }
     }
     catch (UnknownClientException ex)
     {
          MessageBox.Show(ex.Message);
     }
}
```

This action listener performs the following tasks:-

- It opens a dialog box to ask the user for a clients ID. As C# does not contain predefined methods to create input boxes this uses a form called InputBox that was created specifically for this purpose. If cancel is pressed this form returns an empty string.
- It then checks that OK has been pressed and the ID returned is a valid integer.
- On the client book object it invokes the GetClient() method passing the ID as a parameter.
- Assuming a client object is returned, the ToString() method is then invoked to get a string representation of the client and this is passed as a parameter to the MessageBox.Show() method (which displays the details of the client with that ID).
- If GetClient() fails to find a client with the specified ID it will throw an UnknownClientException – this will be caught here and an appropriate message will be displayed. This makes use of the Message property of the Exception class.

11.12 Using Test Driven Development and Extending the System

We now have a working system – though two important methods have yet to be created. We need a method to increase a client's credit – this should be placed within the Client class. We also need a method to delete a client, as this means removing them from the client book. This should be placed in the ClientBook class.

It has been decided to use Test Driven Development to extend the system by providing this functionality (as discussed in Chapter 10 Agile Programming).

In TDD we must :-

1) Write tests
2) Set up automated unit testing, which fails because the classes haven't yet been written!
3) Write the classes so the tests pass

After creating these methods we must then adapt the interface so that this will invoke these methods.

To do this we will create two test fixtures… one to test the ClientBook class and one to test the Client class.

In the Client text fixture we will initialise a test by setting up a specified client. We will then create a test to test credit can be added and finally we will set the TestCleanup() method to remove the client specified.

The code for this is shown below…

```
namespace MessageManagerTests
{
     TestClass]
     public class TestFixture_ClientTests
     {

          private MessageManagerSystem.Clients.Client c;

          [TestInitialize]
          public void TestInitialize()
          {
               c = new MessageManagerSystem.Clients.Client("Simon",
                    "Room 1234", "x200", 10);
          }

          [TestCleanup]
          public void TestCleanup()
          {
               c = null;
          }

          [TestMethod]
          public void IncreaseCredit_TestAdd5UnitsOfCredit
                                                                                    _
CreditShouldBe15()
          {
               c.IncreaseCredit(5);
               Assert.AreEqual(15, c.Credit, "Credit after adding 5
               units is not as expected. Expected: 15 Actual:
               "+c.Credit);
          }

    }
}
```

This test creates a client with 10 units of credit, adds an additional 5 units of credit and then checks that this client has 15 units of credit.

One test does alone not sufficiently prove that the IncreaseCredit() method will always work so we may need to define additional tests.

We also need to create test cases to test the DeleteClient() method in the ClientBook class. As this is a separate class we need to create a new test fixture appropriately named and we need to set up a test to test this method.

The code for this is shown below….

```
namespace MessageManagerTests
{
     [TestClass]
     public class TestFixture_ClientBookTests
     {
          public TestFixture_ClientBookTests()
          {
          }

          private MessageManagerSystem.Clients.ClientBook cb;


          [TestInitialize]
          public void TestInitialize()
          {
               cb = new MessageManagerSystem.Clients.ClientBook();

          }

          [TestCleanup]
          public void TestCleanup()
          {
               cb = null;
          }

          [TestMethod]
          public void GetClient_TestDeleteClient
                                                  _ShouldNotGenerateException()
          {
               MessageManagerSystem.Clients.Client c = new
               MessageManagerSystem.Clients.Client
                                        ("Simon", "Room 1234", "x200", 10);
               try
               {
                    cb.AddClient(1, c);
                    cb.DeleteClient(1);

               }
               catch
               (MessageManagerSystem.Clients.UnknownClientException)
               {
                    Assert.Fail("UnknownClient exception should not be
                                                 thrown if client exists");
               }
          }
```

One test we should perform on the DeleteClient() method is to test that it can delete a client … or at least not generate an exception. The test above proves and exception is not thrown inappropriately but it does not demonstrate that the client has been successfully deleted nor does it test what happens if we try to delete a client thqat does not exist… clearly we need to define some more tests.

Having created appropriate test cases our code will generate complier errors as the methods IncreaseCredit() and DeleteClient() do not exist.

We must now add these methods o our program and revise them until these tests pass.

The IncreaseCredit() method is given below…

```
public void IncreaseCredit(int extraCredit)
{
     credit = credit + extraCredit;
}
```

And the DeleteClient() method is given below…

```
public void DeleteClient(int clientID)
          {
               if(clients.Remove(clientID)==false)
               {
                    throw new
                    UnknownClientException("ClientBook.DeleteClient():
                    unknown client ID:" + clientID);
               }
          }
```

Finally we must amend the system GUI to invoke these methods as required.

Theory suggests that TDD leads to simple code.

In this case by focusing our minds on what the IncreaseCredit() and DeleteClient() methods needs to achieve we reduce the risk of over complicating the code. Of course we may need a range of test cases to make sure the method has all of the essential functionality it needs.

Even if not developing our system using TDD we should define a wide ranging set of test cases for all of the classes within the system. This will ensure that we can undertake regression testing every time we enhance or adapt the system to meet the future and ever changing needs of the client.

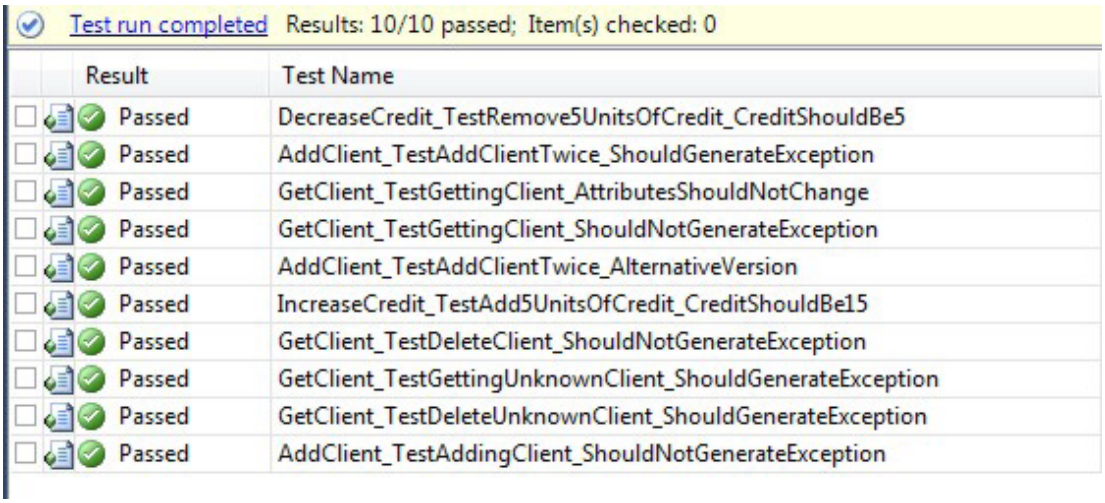
Some more tests for the ClientBook class are shown below….

```
[TestMethod]
public void AddClient_TestAddingClient_ShouldNotGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     try
     {
          cb.AddClient(1, c);
     }
     catch
     (MessageManagerSystem.Clients.ClientAlreadyExistsException)
     {
          Assert.Fail("ClientAlreadyExists exception should not be
          thrown for new clients");
     }

}

[TestMethod]
public void AddClient_TestAddClientTwice_ShouldGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     try
     {
          cb.AddClient(1, c);
          cb.AddClient(1, c);
          Assert.Fail("ClientAlreadyExists exception should be thrown
          if client added twice");
     }
     catch
    (MessageManagerSystem.Clients.ClientAlreadyExistsException)
     {
     }
}

[TestMethod]                              [ExpectedException(typeof(MessageManagerSystem.Clients.
ClientAlreadyExistsException))]
public void AddClient_TestClientTwice_AlternativeVersion()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     cb.AddClient(1, c);
     cb.AddClient(1, c);
     Assert.Fail("ClientAlreadyExists exception should be thrown if
     client added twice");
}

[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException()
{
     try
     {
         cb.GetClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
```

```
     }
     catch(MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}

[TestMethod]
public void GetClient_TestGettingClient_ShouldNotGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     MessageManagerSystem.Clients.Client c2 = null;
     try
     {
          cb.AddClient(1, c);
          c2=cb.GetClient(1);
     }
     catch(MessageManagerSystem.Clients.UnknownClientException)
     {
          Assert.Fail("UnknownClient exception should not be thrown if
          client exists");
     }
}

[TestMethod]
public void GetClient_TestGettingClient_AttributesShouldNotChange()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     MessageManagerSystem.Clients.Client c2 = null;
     try
     {
          cb.AddClient(1, c);
          c2 = cb.GetClient(1);
          Assert.AreEqual(c2.Credit,10,"Value of returned credit not as
          expected");
     }
     catch (MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}


[TestMethod]
public void GetClient_TestDeleteUnknownClient_ShouldGenerateException()
{
     try
     {
          cb.DeleteClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
     }
     Catch (MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}
```

The tests above show numerous tests with an empty client book. They demonstrate that clients can be added, but not twice. They also demonstrate that clients can be deleted and that exceptions are generated appropriately.

The figure below shows the results from running the tests….



By creating automated test fixtures to test all classes and all methods we can run these tests every time the system is adapted to meet the clients changing needs.

11.13 Generating the Documentation

Documentation is essential and can be generated automatically (as described in Chapter 8 - C# Development Tools) assuming appropriate comments have been placed in the code.

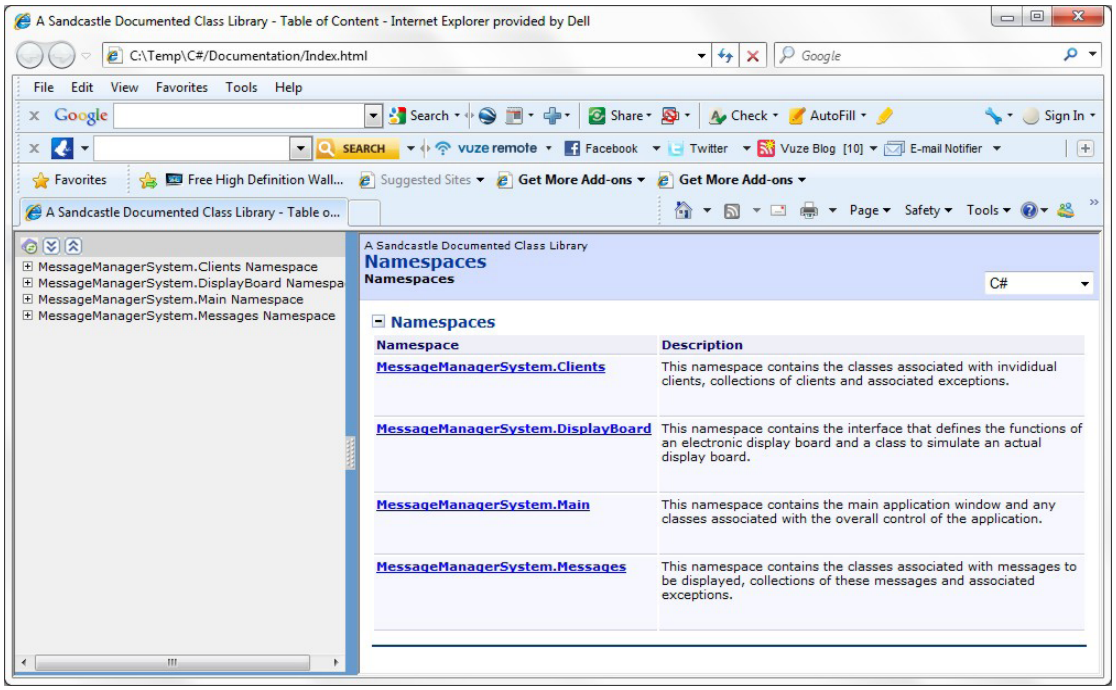
XML comments have been placed in the code to describe all classes, all constructors and all methods. All parameters, return values and exception thrown have also been described.

Three of the comments taken from the Client class are shown below :-

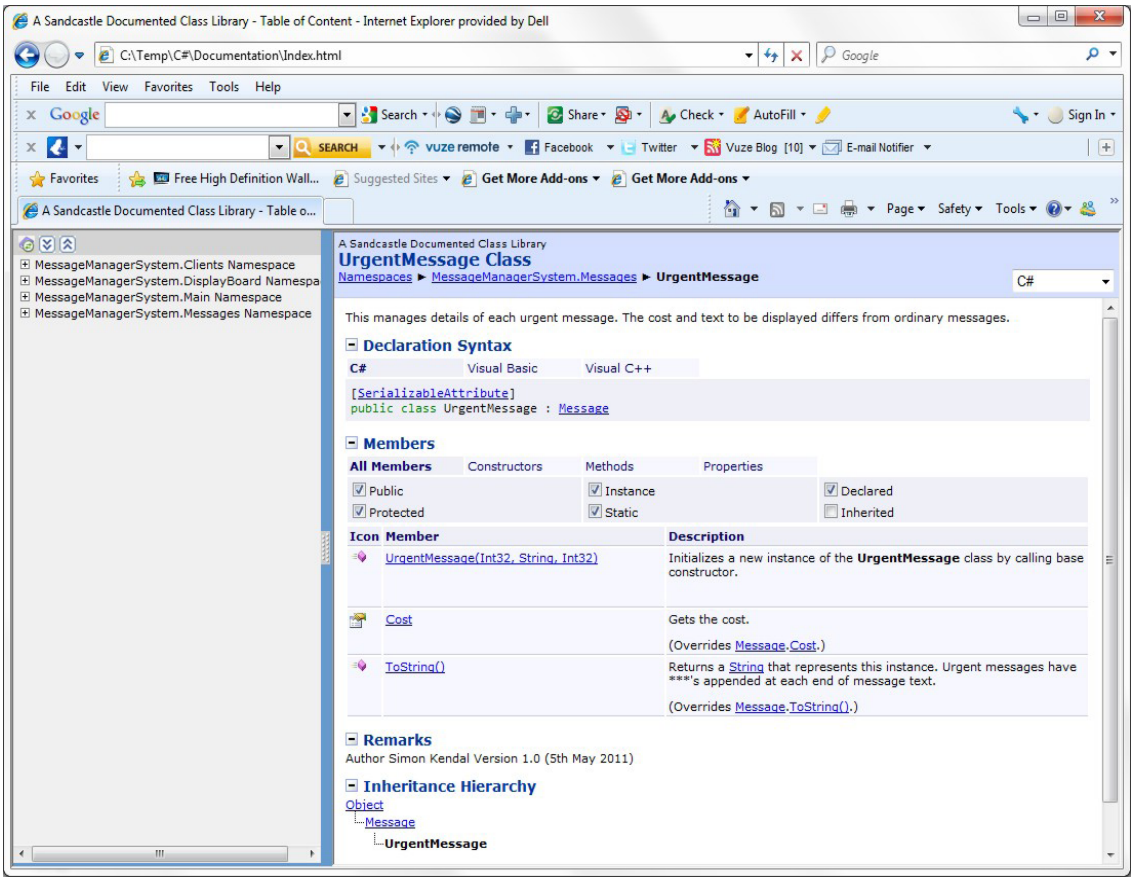
```
/// <summary>
/// Manages a collection (sorted dictionary) of clients where each
/// client has an ID number (int).
/// </summary>
/// <remarks>Author Simon Kendal
/// Version 1.0 (5th May 2011)</remarks>
public class ClientBook
{
      private SortedDictionary<int, Client> clients;

      /// <summary>
      /// Gets the clients.
      /// </summary>
      public  SortedDictionary<int, Client> Clients
      {
           // ... lines missing ...

      }

      /// <summary>
      /// Initializes a new empty instance of the <see
                                                cref="ClientBook"/> class.
      /// </summary>
      public ClientBook()
      {
              // ... lines missing ...
      }

      /// <summary>
      /// Initializes a new instance of the <see cref="ClientBook"/>
                 class and instantiates this to the disctionary passed.
      /// </summary>
      /// <param name="clients">A disctionary of Client ID, Client
                                                              objects.</param>
      public ClientBook(SortedDictionary<int, Client> clients)
      {
           // ... code omitted ...
      }

      /// <summary>
      /// Adds a client to the client book
      /// </summary>
      /// <param name="clientID">The client ID.</param>
      /// <param name="newClient">The new client.</param>
      /// <exception cref="ClientAlreadyExistsException"> Throws
        exception if a client with ClientID already exists</exception>
      public void AddClient(int clientID, Client newClient)
      {
           // ... code omitted ...

      }
```

Once XML comments have been placed throughout the code and exported, and comments have been added to the Sandcastle Help File Builder tool to describe the namespaces then this tool can be used to generate a set of web pages to describe the system…. virtually at the push of a button!

The following picture shows the main help page generated 'Index.html' documentation describing the Message Manager System at its highest most general level i.e. the packages or namespaces within the system.



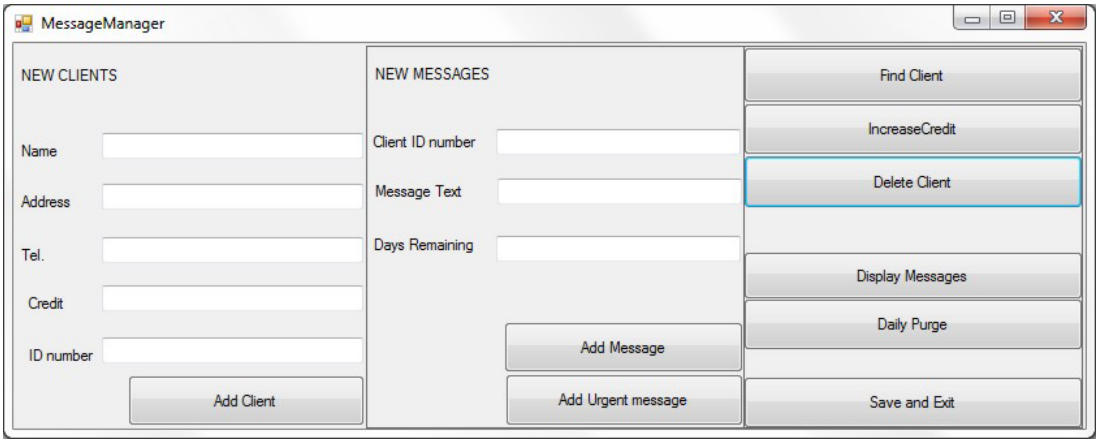
The following picture shows part of the help documentation describing the UrgentMessage class:-



11.14 The Finished System

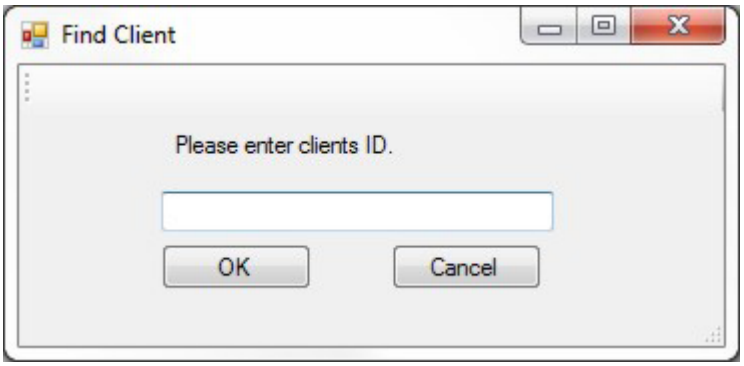
The following screen shots show the finished system.

Firstly the main interface window – this is very similar to the design. The only change was one extra button that was added to allow a message to be designated as an urgent message.

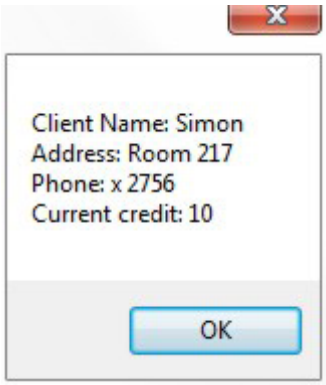


The next two images show the pop up dialogues that appear when the 'Find Client' button is pressed.

Firstly asking for a client ID….



Secondly displaying the client details – assuming a client with this ID has been added.



The 'Display Messages' button shows each of the messages on the screen using the DummyBoard class. This is only crudely simulating a real display board and makes no effort to scroll the messages or display them in any graphically interesting way.

Urgent messages look just like ordinary messages except ***'s are displayed before and after the message.

'Purge Messages' invokes the PurgeMessages() method. Mostly this does nothing visible but decrements the days remaining for each message, decreases the client's credits and deletes the messages if appropriate. Urgent messages are charged at double the rate of ordinary messages. This can be tested by running Find Client before and after doing a daily purge – this should show the clients credit decreasing. If messages exist with an unrecognised client ID and exception will be generated. This exception will be caught by the PurgeMessages() method and an error message will be displayed on the screen.

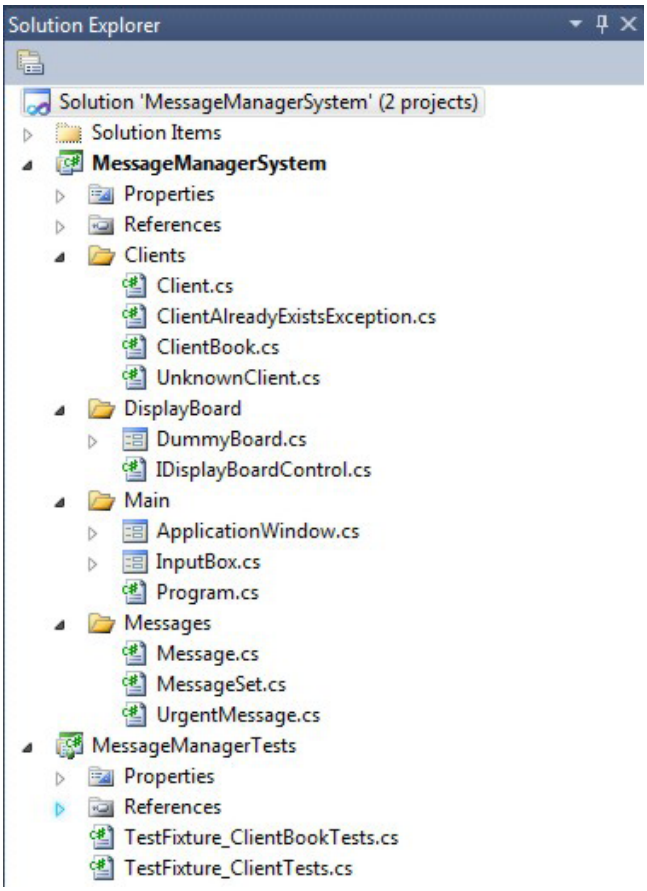
Ultimately of course the idea would be to get the MessageManagerSystem to display the messages on a real display board. This would involve 1) loading the DLL for the real display board 2) creating an object of the real display board in place of the dummy display board 3) passing this object when calling the Display() method. i.e. only two lines of the entire MessageManagerSystem would need to be changed!

11.15 Running the System

The complete, fully commented, source code for the Message Manager system, as described in this chapter, is available with this textbook as zipped file. To view or run the Message Manager system :-

- Install Microsoft Visual Studio 2010 (http://www.microsoft.com/visualstudio/en-us/products/2010-editions). The C# express edition is free and will be perfectly adequate but will not allow you to run the unit tests.
- Download and unzip the file 'OOP Using C#'… available with this book.
- Load the MessageManagerSystem.sln file into Visual Studio, view the code and run within Visual Studio.

In the zip file downloaded are all classes, methods and test cases discussed in this chapter. When viewing the Solution Explorer in Visual Studio you will see all the packages, all of the classes and you should be able to view all of the code with the associated comments (see below..)



You will not able to view to or run the test fixtures with Visual Studio express. Partly to overcome this we have shown many of the test cases in this chapter.

Also inside this zip file is the automated documentation generated by the Sandcastle tool. To view the documentation go to the 'Documentation' folder and double click on the index.html page. This should load the documentation into your web browser software.

If you install Sandcastle Help File Builder (available for free from http://shfb.codeplex.com/) you will be able to load the file MessageManagerSystem.shfbpro available as part of the Message Manager system download. You will then be able to see that the comments for the namespaces have been added to the project properties and if you adjust the output path to an appropriate path for yourself you will be able to rerun this software and see the documentation generated for yourself.

11.6 Conclusions

The fundamental principles of the Object Orientated development paradigm are

- abstraction
- encapsulation
- generalization/specialization (inheritance)
- polymorphism

These principles are ubiquitous throughout the C# language and the .NET APIs as well as providing a framework for our own software development projects.

A well-established range of tools and reference support is available for OO development in C#, some of it allied to modern 'agile' development approaches.

Throughout this chapter you will hopefully have seen how Object Orientation supports the programmer by :-

- using abstraction and encapsulation to enables us to focus on and program different parts of a complex system without worrying about 'the whole'.
- using inheritance to 'factor out' common code
- using polymorphism to make programs easier to change
- using automatic tools to help document and test large software projects.

These principles have been exemplified here using C# but the same principles and benefits apply to all OO programming languages and the facilities demonstrated here are available in many modern IDE's.

Through reading this book, and doing the small exercises, you will hopefully have gained some understanding of these principles.

If you want a further explanation of the C# language the following book is highly recommended…

Pro C# 2010 and the .NET 4 Platform by Andrew Troelsen

Finally I hope you have found this book helpful and I wish you all the best for the future.